conference
proceedings

Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium

Baltimore, MD, USA
August 12–14, 2026

Sponsored by



USENIX
THE ADVANCED COMPUTING SYSTEMS ASSOCIATION

ISBN 978-1-939133-58-8

USENIX
Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium
Baltimore, MD, USA August 12–14, 2026

USENIX Association

Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium

August 12–14, 2026
Baltimore, MD, USA

© 2026 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 978-1-939133-58-8

Artifact Evaluation Committee

Artifact Evaluation Committee Co-Chairs
Aurore Fass, *Inria Centre at Université Côte d'Azur*
Deepak Kumar, *University of California, San Diego*
Christian Wressnegger, *Karlsruhe Institute of Technology*

Artifact Evaluation Publication Chair
Tiago Heinrich, *Max Planck Institute for Informatics*

Artifact Evaluation Committee
Eric Ackermann, *CISPA Helmholtz Center for Information Security*
Shubham Agarwal, *Max Planck Institute for Security and Privacy*
Iman Alipour, *Max Planck Institute for Informatics*
Arshia Arya, *University of California, San Diego*
Md Sadik Awal, *Florida International University*
Sofiane Azogagh, *Université du Québec à Montréal*
Alessandro Baccarini, *Input Output*
Juyang Bai, *Johns Hopkins University*
Weiheng Bai, *University of Minnesota*
Leonhard Balduf, *Technische Universität Darmstadt*
Fabian Bäumer, *Ruhr University Bochum*
Philipp Beer, *TU Wien*
Paschalis Bekos, *Stony Brook University*
Nicola Bottura, *Prisma s.r.l.*
Jiahao Cao, *Tsinghua University*
Katharina Ceesay-Seitz, *ETH Zurich*
M.A.P. Chamikara, *CSIRO's Data61*
Anish Chand, *Louisiana State University*
Hongyan Chang, *Mohamed bin Zayed University of Artificial Intelligence*
Xijia Che, *University of Illinois Urbana–Champaign*
Bocheng Chen, *University of Mississippi*
Boru Chen, *University of California, Berkeley*
Junming Chen, *George Mason University*
Meng Chen, *Zhejiang University*
Simin Chen, *Columbia University*
Yuetian Chen, *Purdue University*
Shao-Yu Chu, *University of California, San Diego*
Paul Chung, *University of California, San Diego*
Jian Cui, *University of Illinois Urbana–Champaign*
Valentino Dalla Valle, *CISPA Helmholtz Center for Information Security*
Vishnu Dasu, *The Pennsylvania State University*
Qing Deng, *University of California, Riverside*
Zizhuang Deng, *Shandong University*
Guillaume Didier, *Saarland University*
Wanning Ding, *Syracuse University*
Weikang Ding, *University of Missouri-Kansas City*
Tian Dong, *The University of Hong Kong*
Jan Drescher, *Technische Universität Braunschweig*
Hassan El Alami, *Howard University*
Zachary Espiritu, *MongoDB Research*
Aurore Fass, *Inria Centre at Université Côte d'Azur*
Fabian Fleischer, *Georgia Institute of Technology*
Marius Fleischer, *NVIDIA*
Jie Fu, *Stevens Institute of Technology*
Julian Funk, *Friedrich-Alexander-Universität Erlangen-Nürnberg*
Eric García Arribas, *TU Graz*
Dañiel Gerhardt, *CISPA Helmholtz Center for Information Security*
Rajdeep Ghosh, *Indian Institute of Technology Kharagpur*
Gwihwan Go, *Tsinghua University*
Bernd Gruner, *German Aerospace Center (DLR)*
Michele Guerra, *New York University Abu Dhabi*
Keno Hassler, *CISPA Helmholtz Center for Information Security*
Lipeng He, *University of Waterloo*
Tiago Heinrich, *Max Planck Institute for Informatics*
Mathé Hertogh, *Vrije Universiteit Amsterdam*
Yiwei Hou, *University of California, Berkeley*
Hongsheng Hu, *University of Newcastle*
Shengtuo Hu, *Palo Alto Networks*
Long Huang, *University of Michigan*
Yujin Huang, *The University of Melbourne*
Michele Ianni, *University of Calabria, Italy*
Abdullah Al Ishtiaq, *The Pennsylvania State University*
Noa Izsak, *CISPA Helmholtz Center for Information Security*
Malvika Jadhav, *University of Florida*
Ismat Jarin, *University of California, Irvine*
Patrick Jattke, *ETH Zurich*
Fujiao Ji, *University of Tennessee, Knoxville*
Zhengjie Ji, *Virginia Tech*
Jinan Jiang, *The Hong Kong Polytechnic University*
Qisheng Jiang, *Duke University*
Yibo Jin, *The Hong Kong University of Science and Technology*
Ehsanul Kabir, *The Pennsylvania State University*
Kelechi Kalu, *Purdue University*
Pratik M. Kamble, *Binghamton University*
Rahul Kande, *Advanced Micro Devices (AMD)*
Ameer Kashani, *DENSO*
Mugdha Khedkar, *Heinz Nixdorf Institute, Paderborn University*
Joseph Khoury, *Louisiana State University*
Dongwook Kim, *Yonsei University*
Jiwon Kim, *Purdue University*
Juhee Kim, *Seoul National University*
Sungwoo Kim, *Purdue University*
Robin Kirchner, *Technische Universität Braunschweig*
Maynard Koch, *Technische Universität Dresden*
Zsolt Kucsvan, *University of Twente*
Ashish Kumar, *The Pennsylvania State University*
Deepak Kumar, *University of California, San Diego*
Seoyoung Kweon, *University of California, San Diego*
Verena Lachner, *University of Innsbruck*
Guilhem Lacombe, *Verimag and Université Grenoble-Alpes*
Felix Lange, *Paderborn University*
Jinseo Lee, *Korea Advanced Institute of Science and Technology (KAIST)*
Seungju Lee, *Princeton University*
Ahmed Lekssays, *Qatar Computing Research Institute*
Jiate Li, *University of Southern California*
Kunlin Li, *The Hong Kong University of Science and Technology (Guangzhou)*

Kunyang Li, *University of Wisconsin—Madison*
Peiyang Li, *Ant Group and Tsinghua University*
Xiang Li, *George Mason University*
Ying Li, *University of California, Los Angeles*
Youpeng Li, *The University of Texas at Dallas*
Junkai Liang, *Peking University*
Kyungchan Lim, *University of Maryland, College Park*
Elizabeth Lin, *North Carolina State University*
Zhengyao Lin, *Carnegie Mellon University*
Zilong Lin, *University of Missouri-Kansas City*
Changming Liu, *Google*
Junming Liu, *Max Planck Institute for Security and Privacy*
Junrui Liu, *University of California, Santa Barbara*
Mengling Liu, *The Hong Kong Polytechnic University*
Mingyi Liu, *Georgia Institute of Technology*
Puzhuo Liu, *Ant Group and Tsinghua University*
Side Liu, *Tulane University*
Xuanming Liu, *Zhejiang University*
Yichen Liu, *University of Illinois Urbana–Champaign*
Yupei Liu, *The Pennsylvania State University*
Minjun (Elena) Long, *University of Virginia*
Feng Luo, *The Hong Kong Polytechnic University*
Kaixuan Luo, *The Chinese University of Hong Kong*
Muxi Lyu, *University of California, Berkeley*
Xingyu Lyu, *University of Massachusetts Lowell*
Binhao Ma, *University of Missouri-Kansas City*
Jie Ma, *Beihang University*
Kai Ma, *Huazhong University of Science and Technology*
Xiaoyue Ma, *George Mason University*
Zuchao Ma, *The Hong Kong Polytechnic University*
Marcel Maehren, *Ruhr University Bochum*
Noah Mauthe, *CISPA Helmholtz Center for Information Security*
Federico Mazzone, *University of Twente*
Yassine Mekdad, *Florida International University*
Sepehr Mirzaei, *Saarland University*
Atefeh Mohseni-Ejiyeh, *University of California, Santa Barbara, and Harness*
Hamza Mouhcine, *Qatar Computing Research Institute*
Jonas Mücke, *Technische Universität Dresden*
Antonio Muñoz, *University of Malaga*
Nima Naderloui, *University of Connecticut*
Hocheol Nam, *Korea Advanced Institute of Science and Technology (KAIST)*
Kevin Nam, *Seoul National University*
Tushar Nayan, *Florida International University*
Nurlan Nazaraliyev, *University of California, Riverside*
Ravan Nazaraliyev, *University of California, Riverside*
Niklas Niere, *Paderborn University*
Jean-Charles Noirot Ferrand, *University of Wisconsin—Madison*
Paul Olivier, *LAAS-CNRS*
Fariba Osali, *Max Planck Institute for Informatics*
Yumingzhi Pan, *Southeast University*
David Pape, *CISPA Helmholtz Center for Information Security*
Giuseppe Parrella, *University of Salerno*
Huiyun Peng, *Purdue University*
Zifan Peng, *The Hong Kong University of Science and Technology (Guangzhou)*
Daniel Pöllmann, *ETH Zurich and Lucerne University of Applied Sciences and Arts*
Juefei Pu, *University of California, Riverside*
Siliang Qin, *Institute of Information Engineering, Chinese Academy of Sciences*
Vaishnavi Raghavajosyula, *Max Planck Institute for Informatics*
Shoupeng Ren, *Zhejiang University*
Arman Riasi, *Virginia Tech*
Mohamadreza Rostami, *Technische Universität Darmstadt*
Florian Roudot, *University of Rennes*
Bonan Ruan, *National University of Singapore*
Amit Samanta, *University of Utah*
Sambhav Satija, *University of Wisconsin—Madison*
Shayan Shabihi, *University of Maryland, College Park*
Leming Shen, *The Hong Kong Polytechnic University*
Zewei Shi, *The University of Melbourne and CSIRO's Data61*
Ye Shu, *University of California, San Diego*
Sachin Shukla, *Microsoft*
Sachin Kumar Singh, *University of Utah*
Amit Singha, *Purdue University*
Devansh Srivastav, *CISPA Helmholtz Center for Information Security*
Tim Leonhard Storm, *Paderborn University*
Tong Sun, *Zhejiang University*
Guofeng Tang, *Singapore Management University*
Fabian Thomas, *CISPA Helmholtz Center for Information Security*
Yunpeng Tian, *The Hong Kong Polytechnic University*
Dominic Troppmann, *CISPA Helmholtz Center for Information Security*
Yu-Jye Tung, *The Pennsylvania State University*
Mughees Ur Rehman, *Virginia Tech*
Corban Villa, *University of California, Berkeley*
Arturo Villacañas, *Independent Researcher*
Guruprasad Viswanathan Ramesh, *University of Wisconsin—Madison*
Ryan Vrecenar, *Sandia National Laboratories*
Chenkai Wang, *University of Illinois Urbana–Champaign*
Chenyi Wang, *University of Arizona*
Lihua Wang, *University of New South Wales*
Longxiang Wang, *City University of Hong Kong*
Yaofei Wang, *Hefei University of Technology*
Zihan Wang, *The University of Queensland and CSIRO's Data61*
Thom Wiggers, *PQShield*
Annika Wilde, *Ruhr University Bochum*
Luca Wilke, *Microsoft Research*
Christian Wressnegger, *Karlsruhe Institute of Technology*
Stanley Wu, *University of Chicago*
Yuwei Xiao, *LinkedIn*
Zhikang Xie, *The University of Hong Kong*
Peiran Xu, *University of California, Los Angeles*
Tianshi Xu, *Peking University*
Ziqing Yang, *CISPA Helmholtz Center for Information Security*
Shenglong Yao, *Georgia Institute of Technology*
Taeung Yoon, *University of Illinois Urbana–Champaign*
Zi-Quan You, *Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University*
Jiahao Yu, *Northwestern University*

Jiangrui Yu, *Peking University*
Jiongchi Yu, *Singapore Management University*
Jiping Yu, *Tsinghua University*
Mehtab Zafar, *National University of Singapore*
Mingming Zha, *Indiana University Bloomington*
Xiao Zhan, *Universitat Politècnica de València and University of Cambridge*
Bo Zhang, *Northwestern Polytechnical University*
Bosi Zhang, *Huazhong University of Science and Technology*
Junqi Zhang, *University of Science and Technology of China*
Shenyi Zhang, *Wuhan University*
Tingwei Zhang, *Cornell Tech*
Xiang Zhang, *Tianjin University*
Xiaohan Zhang, *Fudan University*
Xin Zhang, *Peking University*
Zhiyu Zhang, *Institute of Information Engineering, Chinese Academy of Sciences*
Zhou Zhang, *Beihang University*
Kunsong Zhao, *The Hong Kong Polytechnic University*
Yao Zhao, *Victoria University*
Xia Zhou, *Zhejiang University*
Yi Zhou, *Amazon Web Services*
Jinhao Zhu, *University of California, Berkeley*
Zeyang Zhuang, *The Chinese University of Hong Kong*
Haodi Zou, *University of California, San Diego*
Yunkai Zou, *Nankai University*

Message from the USENIX Security '26 Artifact Evaluation Committee Co-Chairs

On behalf of USENIX, we are pleased to welcome you to the Artifact Appendices to the Proceedings of the 35th USENIX Security Symposium. This year marks the second year of artifact evaluation as a mandatory part of the conference under USENIX Security's Open-Science policy. Since its introduction in 2020, artifact evaluation has steadily grown in both scope and adoption, with the transition to mandatory artifact evaluation representing an important milestone in the maturation of this process. Building on the experience gained during the first year under this policy, our collective goal remains the same: to strengthen the reproducibility, transparency, and impact of security research.

Following the process introduced last year, artifact evaluation is organized into two distinct phases. The first phase, conducted after paper acceptance, focuses on mandatory availability checks, ensuring that all artifacts referenced in accepted papers are publicly and permanently accessible online. At the end of this phase, authors submit the camera-ready version of their paper with a link to the available artifact. The second phase, following the final paper submission, optionally evaluates artifact functionality and reproducibility. This phase is based on an artifact appendix that describes how to use the artifact and, in particular, how to reproduce the main claims of the accompanying paper.

The two-phase structure allows us to enforce the Open-Science policy while providing authors and evaluators with the time and structure needed to assess artifacts thoroughly. As with paper submissions, the artifact evaluation process offers two submission cycles, each consisting of the two phases described above.

The result was a number of 350 artifacts formally reviewed, reflecting both the enthusiasm and the challenges of mandatory participation. We saw a wide diversity of artifacts—ranging from survey instruments to full experimental frameworks—across all security topics. Each artifact received at least two reviews for phase 1 (availability verification) and an additional two for phase 2 (functionality and reproducibility assessments). The review process for availability verification took 2–3 weeks, and five weeks for functionality and reproducibility assessments. During this review period, authors and reviewers interacted to solve any problems, clarify some instructions, and overall improve the quality of the artifacts; this discussion phase led to more than 2,390 comments. Throughout, the Artifact Evaluation Committee (AEC) demonstrated remarkable diligence in applying the badge criteria: Artifacts Available, Artifacts Functional, and Results Reproduced. To build a strong AEC, we reached out to the community, past AEC members of the major security conferences, and authors of accepted papers at USENIX Security '25, to (self-)nominate candidates.

During artifact evaluation, each evaluator reviewed on average two artifacts per cycle (minimum of one, maximum of eight for AEC members who volunteered for more). In parallel to the reviews, the artifacts and appendices were automatically checked to ensure that:

1. authors provided a stable reference to the stored artifacts; and
2. that the appendices were complete and formatted correctly. This helps to guarantee reproducibility of research results through reliable artifacts.

In total, 350 papers (155 in Cycle 1 and 195 in Cycle 2) participated in the mandatory availability verification, and all were awarded the Artifacts Available (AA) badge for making their research artifacts permanently accessible. Ultimately, 130 papers received the Artifacts Functional (AF) badge, recognizing the successful execution, documentation, and completeness of their artifacts, and 105 papers earned the Results Reproduced (RR) badge, demonstrating that their artifacts could be used with the provided documentation to reproduce the experimental results. Congratulations to all!

To further promote excellence in artifact evaluation and highlight outstanding contributions, we continued the tradition of presenting awards to both authors and evaluators. This year's Distinguished Artifact Awards were determined through votes from the entire Artifact Evaluation Committee (AEC). Five papers were selected for this honor (https://secartifacts.github.io/usenixsec2026/awards#-distinguished-artifact-awards), each earning near-unanimous praise for artifacts that reproduced all claims out of the box, included comprehensive documentation and automation, and offered responsive author support. Most importantly, these artifacts are expected to have a broad impact and inspire future research in our community.

The Distinguished Reviewer Awards (11 recipients, https://secartifacts.github.io/usenixsec2026/awards#-distinguished-reviewer-awards) were given to evaluators who went above and beyond by providing exceptionally detailed, constructive feedback that not only strengthened the evaluation process but, in many cases, also improved the accompanying papers. The Noteworthy Reviewers Recognition (33 recipients, https://secartifacts.github.io/usenixsec2026/awards#-noteworthy-reviewer-recognition) recognized committee members who consistently delivered clear, thoughtful, and timely reviews, demonstrating reliability and engagement throughout the evaluation. Finally, the Artifact Ninja Reviewers Recognition honored 13 recipients (https://secartifacts.github.io/usenixsec2026/awards#-ninja-reviewer-recognition) who stepped in to take on additional reviews, often at short notice, to ensure timely and thorough evaluations when unexpected challenges arose.

Together, these awards and recognitions celebrate the collective effort of the AEC and the broader community, acknowledging both the authors who set a high bar for artifact quality and the reviewers whose dedication ensured a fair, rigorous, and supportive evaluation process.

We are tremendously grateful to the AEC. The evaluators spent countless hours diligently working with the authors to help them improve their artifacts. Without this effort, the artifact verification would not have been possible.

Aurore Fass, *Inria Centre at Université Côte d'Azur*
Deepak Kumar, *University of California, San Diego*
Christian Wressnegger, *Karlsruhe Institute of Technology*
USENIX Security '26 Artifact Evaluation Committee Co-Chairs

Tiago Heinrich, *Max Planck Institute for Informatics*
USENIX Security '26 Artifact Evaluation Publication Chair

35th USENIX Security Symposium

August 12–14, 2026
Baltimore, MD, USA

Wednesday, August 12

Applications of Cryptography 1

Orbit: Optimizing Rescale and Bootstrap Placement with Integer Linear Programming Techniques for Secure Inference 1
Zikai Zhou, *Tsinghua University;* William Seo and Edward Chen, *Carnegie Mellon University;* Alex Ozdemir, *Max Planck Institute for Security and Privacy;* Fraser Brown and Wenting Zheng, *Carnegie Mellon University*

Libra: Pattern-Scheduling Co-Optimization for Cross-Scheme FHE Code Generation over GPGPU 5
Song Bian, Yintai Sun, Zian Zhao, and Haowen Pan, *Beihang University;* Mingzhe Zhang, *unaffiliated;* Zhenyu Guan, *Beihang University*

VROOM: Accelerating (Almost All) Number-Theoretic Cryptography Using Vectorization and the Residue Number System 9
Simon Langowski, Kaiwen He, and Srinivas Devadas, *MIT*

Bridging Usability and Performance: A Tensor Compiler for Autovectorizing Homomorphic Encryption 15
Edward Chen, Fraser Brown, and Wenting Zheng, *Carnegie Mellon University*

Security of ML 1

Fend for Yourself! Backdoor Purification in Federated Graph Learning with an Evolving Knowledge Anchor 19
Chengcheng Zhu and Yunlong Mao, *Nanjing University;* Jiale Zhang and Bosen Rao, *Yangzhou University;* Sheng Zhong, *Nanjing University*

Network Security 1

Breaking the Boundaries: Analyzing QUIC Frame-Packet Interactions With QUIC-Attacker 23
Nurullah Erinola, Marcel Maehren, Marcus Brinkmann, and Jörg Schwenk, *Ruhr University Bochum*

Secure Protocol Composition under Dynamic Corruption: Scaling Up Symbolic Analysis for Real-World Security Properties 27
Cas Cremers, Erik Pallas, and Aleksi Peltonen, *CISPA Helmholtz Center for Information Security*

Semantics Over Syntax: Uncovering Pre-Authentication 5G Baseband Vulnerabilities 31
Qiqing Huang and Xingyu Wang, *University at Buffalo;* Wanda Guo and Guofei Gu, *Texas A&M University;* Hongxin Hu, *University at Buffalo*

Hardware Security 1

Principled Design of Indexing Functions for Memory Coloring 37
Stephan Dübler and Jana Hofmann, *Max Planck Institute for Security and Privacy* (*MPI-SP*)*;* Boris Köpf and Stavros Volos, *Azure Research, Microsoft*

An Integer Overflow Endgame: Compiler and Architecture Support for Default-On Integer Overflow Mitigation 41
Zheng Zhang, Qi Ling, and Kian Kasad, *Purdue University;* Brendan Ryan Sweeney, *UT Austin;* Deepak Gupta, *Meta Platforms Inc.;* Tal Garfinkel, *Google;* Kazem Taram, *Purdue University*

Khost: KVM-based Near Native MCU Firmware Rehosting 43
Chunlin Wang, Yicheng Yang, Yuan Zhang, Haoyu Xiao, Yifan Zhang, and Jiarun Dai, *Fudan University*

Security and Privacy for Web, Mobile, and Emerging Technologies 1

When HTTP 402 Meets the Blockchain: Risks on Emerging x402 Payments 47
Qinying Wang, *EPFL;* Yong Yang, *Zhejiang University;* Yuan Chen, *Independent Researcher;* Shouling Ji, *Zhejiang University;* Mathias Payer, *EPFL*

System Security 1

KernelRCA: Facilitating Root Cause Analysis of Memory Corruptions in Linux Kernel with Contextual Causality Chain **53**
Kangzheng Gu, Yifan Zhang, Yuan Zhang, and Min Yang, *Fudan University*

Discovering, characterizing and exploiting controllable-copy objects for kernel data-only attacks with CopyKat **57**
Jakob Koschel, Andrea Mambretti, Alessandro Sorniotti, Pietro Moretto, and Claudio Migliorelli, *IBM Research Europe;* Andrea Di Dio and Cristiano Giuffrida, *Vrije Universiteit Amsterdam;* Anil Kurmus, *IBM Research Europe*

SoK: On the Fragility of Memory Error Exploit Mitigations **63**
Adriaan Jacobs, *DistriNet, KU Leuven;* Mahmoud Ammar, *Huawei Research, Germany;* Stijn Volckaert, *DistriNet, KU Leuven*

kSFS: Repurposing a Microkernel-like Interface for Fast and Secure In-Kernel Linux File Systems **67**
Dinglan Peng and Pedro Fonseca, *Purdue University*

DMGuard: Safeguarding Kernels from Physical-Page Use-After-Free Vulnerabilities **69**
Juhee Kim, Jaeyoung Chung, Dae R. Jeong, and Byoungyoung Lee, *Seoul National University*

Applications of Cryptography 2

Bridging Bitcoin to Second Layers via BitVM2 . . . . . **71**
Robin Linus Woll, *Stanford University and ZeroSync Association;* Lukas Aumayr, *University of Edinburgh and Common Prefix;* Zeta Avarikioti, *TU Wien and Common Prefix;* Matteo Maffei, *TU Wien;* Andrea Pelosi, *University of Pisa, University of Camerino, and TU Wien;* Orfeas Stefanos Thyfronitis Litos, *Imperial College London and Common Prefix;* Christos Stefo, *TU Wien;* David Tse, *Stanford University and Byzantine Research;* Alexei Zamyatin, *BOB*

Privacy of ML 1

Can we estimate privacy vulnerability of individual records? Towards Mitigating Attribute Inference Attacks on ML Models **75**
Ehsanul Kabir and Najrin Sultana, *Pennsylvania State University;* Ninghui Li, *Purdue University;* Shagufta Mehnaz, *Pennsylvania State University*

Five Queries Are Enough: Query-Efficient and Surrogate-Free Membership Inference Attacks on RAG via Entailment **79**
Nguyen Linh Bao Nguyen, Wanlun Ma, and Viet Vo, *Swinburne University of Technology, Australia;* Alsharif Abuadbba, *CSIRO, Australia;* Minghong Fang, *University of Louisville, USA;* Jun Zhang and Yang Xiang, *Swinburne University of Technology, Australia*

Membership Inference Attacks on Tokenizers of Large Language Models **83**
Meng Tong, *University of Science and Technology of China;* Yuntao Du, *Purdue University;* Kejiang Chen and Weiming Zhang, *University of Science and Technology of China;* Ninghui Li, *Purdue University*

Privacy Anonymity 1

Distributed Synthesis of Differentially Private Tabular Datasets **85**
Yucheng Fu, *University of Virginia;* Tianyao Gu and Elaine Shi, *Carnegie Mellon University;* Tianhao Wang, *University of Virginia*

The Adverse Effects of Omitting Records in Differential Privacy: How Sampling and Suppression Degrade the Privacy–Utility Tradeoff **89**
Àlex Miranda-Pascual, *Kalrsruhe Institute of Technology and Universitat Politècnica de Catalunya;* Javier Parra-Arnau, *Universitat Politècnica de Catalunya;* Thorsten Strufe, *Karlsruhe Institute of Technology*

High-Accuracy, Poisoning-Resilient Frequency Estimation in the Shuffle Model **93**
Shaoqiang Wu, *CS&CCS, DISSec, AAIS, Nankai University;* Jingyu Jia, *CS&CCS, DISSec, AAIS, Nankai University; Automotive New Technology Research Institute of BYD;* Yikuan Zhu and Xinhao Li, *CS&CCS, DISSec, AAIS, Nankai University;* Changyu Dong, *Guangzhou University;* Zheli Liu, *CS&CCS, DISSec, AAIS, Nankai University*

Interpolation-Based Optimization for Enforcing *lp*-Norm Metric Differential Privacy in Continuous and Fine-Grained Domains 97
Chenxi Qiu, *University of North Texas*

Human Aspects 1

Love, Lies, and Language Models: Investigating AI's Role in Romance-Baiting Scams 101
Gilad Gressel and Rahul Pankajakshan, *Center for Cybersecurity Systems & Networks, Amrita Vishwa Vidyapeetham, Amritapuri;* Shir Rozenfeld, *Ben Gurion University of the Negev;* Ling Li, *Ca' Foscari University of Venice;* Ivan Franceschini, *University of Melbourne;* Krishnashree Achuthan, *Center for Cybersecurity Systems & Networks, Amrita Vishwa Vidyapeetham, Amritapuri;* Yisroel Mirsky, *Ben Gurion University of the Negev*

"Do Not Mention This to the User": Detecting and Understanding Malicious Agent Skills in the Wild 105
Yi Liu, Zhihao Chen, and Yanjun Zhang, *Griffith University;* Gelei Deng, *Nanyang Technological University;* Yuekang Li, *University of New South Wales;* Jianting Ning, *Zhejiang Key Laboratory of Digital Fashion and Data Governance, Zhejiang Sci-Tech University;* Leo Yu Zhang, *Griffith University*

Software Security 1

Exposing Resource-Exhaustion DoS Vulnerabilities with Leak-Oriented Minimum Path Covers 109
Lige Zhan and Yafei He, *Wuhan University;* Jiang Ming, *Tulane University;* Guojun Peng and Jianming Fu, *Wuhan University*

Thursday, August 13

Privacy Anonymity 2

United We Defend: Collaborative Membership Inference Defenses in Federated Learning 113
Li Bai, *The Hong Kong Polytechnic University;* Junxu Liu, *The Hong Kong Polytechnic University and PolyU Research Centre for Privacy and Security Technologies in Future Smart Systems;* Sen Zhang, Xinwei Zhang, and Qingqing Ye, *The Hong Kong Polytechnic University;* Haibo Hu, *The Hong Kong Polytechnic University and PolyU Research Centre for Privacy and Security Technologies in Future Smart Systems*

Security of ML 3

What the Eyes See, the LLMs Miss: Exploiting Human Perception for Adversarial Text Attacks 115
Qin Yang, *University of Connecticut;* Lu Malloy, *University of Tennessee;* Joshua Lee, *University of California, Santa Barbara;* Xiaohan Chang, *University of Connecticut;* Meisam Mohammady, *Iowa State University;* Doowon Kim, *University of Tennessee;* Yuan Hong, *University of Connecticut*

Network Security 2

A Formal Security Analysis of CAN XL 117
Zhaozhou Tang, *Georgia Institute of Technology;* Khaled Serag, *Qatar Computing Research Institute;* Z. Berkay Celik, *Purdue University;* Vijay Ganesh, Saman Zonouz, and Raheem Beyah, *Georgia Institute of Technology*

SoK: The Pitfalls of Deep Reinforcement Learning for Cybersecurity 121
Shae McFadden, *King's College London, The Alan Turing Institute, and University College London;* Myles Foley, *The Alan Turing Institute and Devotion AI Labs;* Elizabeth Bates, *The Alan Turing Institute;* Ilias Tsingenopoulos, *KU Leuven;* Sanyam Vyas, *Cardiff University and The Alan Turing Institute;* Vasilios Mavroudis, *King's College London;* Chris Hicks, *The Alan Turing Institute;* Fabio Pierazzi, *University College London*

Hardware Security 2

InstrSem: Automatically and Generically Inferring Semantics of (Undocumented) CPU Instructions 125
Lorenz Hetterich, Fabian Thomas, Tristan Hornetz, and Michael Schwarz, *CISPA Helmholtz Center for Information Security*

HAMLOCK: HArdware-Model LOgically Combined attacK 127
Sanskar Amgain, *University of Tennessee, Knoxville;* Daniel Lobo, Atri Chatterjee, and Swarup Bhunia, *University of Florida;* Fnu Suya, *University of Tennessee, Knoxville*

Loongleak: Architectural Cross-Privilege-Boundary Data Leakage on LoongArch CPUs 131
Lorenz Hetterich, Tristan Hornetz, Fabian Thomas, and Michael Schwarz, *CISPA Helmholtz Center for Information Security*

Fuzzing Open-Source GPU Hardware with SIMT Program Generation **135**
Zibo Gao, Jie Wang, Qihang Zhou, Lixiao Shan, Junjie Hu, Xiaoqi Jia, and Zhiqiang Lv, *Institute of Information Engineering, Chinese Academy of Sciences; School of Cyber Security, University of Chinese Academy of Sciences*

Injected and Leaked: Actively Inducing Side-Channel Leakage Using Electromagnetic Injection and Hardware Nonlinearity **137**
Haoran Yan, Ziyu Shao, and Shuhao Zhang, *The Hong Kong University of Science and Technology* (*Guangzhou*); Qinhong Jiang, *The Hong Kong Polytechnic University;* Yan Long, *The Hong Kong University of Science and Technology* (*Guangzhou*)

***DRVFuzz*: Data-Sensitive RISC-V CPU Fuzzing** **141**
Zehong Yu, *Tsinghua University;* Yuanliang Chen, *Renmin University of China;* Zhen Yan, Xudong Zhang, Zhensheng Xian, and Yu Jiang, *Tsinghua University*

Security and Privacy for Web, Mobile, and Emerging Technologies 2

"Tap" Without Tapping: A Tag Discovery Forgery Attack on Android NFC **145**
Yilin Li, *Shandong University;* Jianliang Wu, *Simon Fraser University;* Chaoshun Zuo, *Independent Researcher;* Qingchuan Zhao, *City University of Hong Kong;* Xiaofeng Liu, Xiangpu Song, Chengyu Hu, and Shanqing Guo, *Shandong University*

AutoFail: Breaking Web Boundaries using Android's Autofill Framework **149**
Riccardo Lamarca, Philipp Beer, and Marco Squarcina, *TU Wien*

Plain Text, Plain Risks: Measuring HTTP Inclusion in Android WebViews at Scale **153**
Philipp Beer, *TU Wien;* Sebastian Roth, *University of Bayreuth;* Martina Lindorfer and Marco Squarcina, *TU Wien*

Software Security 2

Leveraging Cryptographic Simulator Synthesis for Formally Verifying the FOO E-Voting Protocol **157**
David Baelde, *Univ Rennes, CNRS, IRISA;* Adrien Koutsos and Justine Sauvage, *Inria*

OS-Sanitizer: System-wide Latent Defect Inference in Linux Applications **161**
Addison Crump, Sahil Sihag, Florian Bauckholt, and Keno Hassler, *CISPA Helmholtz Center for Information Security;* Thorsten Holz, *Max Planck Institute for Security and Privacy*

Melting the Flesh of PHP's Memory Hardening **165**
Yifan Wu, *University of California, Riverside;* Xiaochuan Yu, *University of California, San Diego;* Zhiyun Qian, *University of California, Riverside*

Retrofit: Continual Learning with Controlled Forgetting for Binary Security Detection and Analysis **169**
Yiling He, *University College London;* Junchi Lei, Hongyu She, and Shuo Shao, *Zhejiang University;* Xinran Zheng, *University College London;* Yiping Liu, *Alibaba Group;* Zhan Qin, *Zhejiang University;* Lorenzo Cavallaro, *University College London*

SafeFFI: Efficient Sanitization at the Boundary Between Safe and Unsafe Code in Rust and Mixed-Language Applications **173**
Oliver Braunsdorf and Tim Lange, *Ludwig-Maximilians-Universität München;* Konrad Hohentanner and Julian Horsch, *Fraunhofer AISEC;* Johannes Kinder, *Ludwig-Maximilians-Universität München*

Applications of Cryptography 3

Enhanced Private Set Union from Secret-shared Private Membership Test **179**
Meng Hao, *Singapore Management University;* Guodong Wang, *University of Electronic Science and Technology of China;* Xinpeng Yang, *Nanyang Technological University;* Pengzhi Xing and Hanxiao Chen, *University of Electronic Science and Technology of China;* Tianwei Zhang, *Nanyang Technological University;* Haiyang Xue and Guomin Yang, *Singapore Management University;* Hongwei Li, *University of Electronic Science and Technology of China;* Robert H. Deng, *Singapore Management University*

SING: Improving the Efficiency of MPC Protocol Assignment using Graph Neural Networks **183**
Jannis Blüml, Moritz Huppert, Nora Khayata, Joachim Schmidt, and Thomas Schneider, *Technical University of Darmstadt*

PICS: Private Intersection over Committed (and reusable) Sets 189
Aarushi Goel, *Rutgers University;* Peihan Miao and Phuoc Van Long Pham, *Brown University;* Satvinder Singh, *Purdue University*

FLOSS: Fast Linear Online Secret-Shared Shuffling 193
Ian Chang and Sela Navot, *University of Washington;* Alex Ozdemir, *Max Planck Institute for Security & Privacy, Georgia Tech;* Nirvan Tyagi, *University of Washington*

Privacy Anonymity 3

LPG: Raise Your Location Privacy Game in Direct-to-Cell LEO Satellite Networks. 199
Quan Shi, *National University of Singapore;* Liying Wang, *Peking University;* Prosanta Gope, *University of Sheffield;* Qi Liang and Haowen Wang, *Beijing University of Posts and Telecommunications;* Qirui Liu, *Peking University;* Chenren Xu, *Peking University and Key Laboratory of High Confidence Software Technologies, Ministry of Education (PKU);* Shangguang Wang and Qing Li, *Beijing University of Posts and Telecommunications;* Biplab Sikdar, *National University of Singapore*

***Revelio*: A Global Study of VoIP Blocking 203**
Arjun Gupta, *IIT Bombay;* Roya Ensafi, *University of Michigan;* Piyush Kumar, *IIT Delhi;* Devashish Gosain, *IIT Bombay*

PrivacyShield: Relaying BLE Beacons to Counter Unsolicited Tracking 207
Florian Hofhammer, *EPFL;* Daniele Antonioli, *EURECOM;* Mathias Payer, *EPFL*

Privacy Anonymity 4

Do You Need a Receipt? Anonymous Credential Revocation at Continental Scale via Private Record Certification . . . 211
Kasra EdalatNejad, Sebastian Faust, Jonas Hofmann, Philipp-Florens Lehwalder, and Thomas Schneider, *Technical University of Darmstadt*

Remise: Authorized Anonymous Communication Systems 215
Rohan Ravi, Paritosh Shukla, and Adithya Vadapalli, *IIT Kanpur*

SophOMR: Improved Oblivious Message Retrieval from SIMD-Aware Homomorphic Compression. 221
Keewoo Lee, *Ethereum Foundation;* Yongdong Yeo, *Seoul National University*

sigma-rs: A Modular Approach for Keyed-Verification Anonymous Credentials 223
Michele Orrù, *CNRS;* Lindsey Tulloch, *Tor Project, University of Waterloo;* Victor Snyder-Graf, *RISC Zero;* Ian Goldberg, *University of Waterloo*

Hardware Security 3

FIRA: Enabling Automatic Forensic Investigation of Unmanned Aerial Vehicles 227
Yizhi Huang, *Georgia Institute of Technology;* David Oygenblik, *Georgia Tech;* Runze Zhang, Mingxuan Yao, Muhammad Ibrahim, Burak Sahin, Haichuan Xu, Saman Zonouz, and Brendan Saltaformaggio, *Georgia Institute of Technology*

Arto: Efficient Execution Integrity Attestation for Real-Time Operation of Cyber-Physical Systems. 231
Ruizhe Zhao, Cong Sun, and Zongzhen Li, *State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University;* Tiantian Wang, *Northeastern University;* Yunbo Wang, *State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University*

BLE Theft Auto: Evaluating the Security of Aftermarket BLE-based Automotive Remote Control Systems 235
Jerry Yu, Yibo Wei, Sumanth Rao, Mohak Vaswani, Jefferson Chien, Christian Dameff, Nishant Bhaskar, and Aaron Schulman, *UC San Diego*

Network Security 3

Hop: A Modern Transport and Remote Access Protocol 239
Paul Flammarion, *Stanford University;* George Hosono, *Georgia Institute of Technology;* Wilson Nguyen, Laura Bauman, Daniel Rebelsky, and Gerry Wan, *Stanford University;* David Adrian, *Independent;* Zakir Durumeric, *Stanford University*

WAVED: Principled Identification of Off-Path Exploitable Weak Verifications within the TCP/IP Protocol Suite . . . 243
Yizhou Zhao and Xuewei Feng, *Tsinghua University;* Min Li, *Zhongguancun Laboratory;* Ke Xu, *Tsinghua University*

System Security 2

Download More RAM: Dismantling Windows Operating System Defences with Mischievous Memory 245
Sam Collins, Tom Chothia, William Burgess, and Marius Muench, *The University of Birmingham;* David Oswald, *Durham University*

Secpoline: A Scalable Approach to Build Secure In-Process Syscall Interposers 249
Ruben Sturm and Anton Schelfhout, *DistriNet, KU Leuven;* Merve Gülmez, *Ericsson Security Research;* Adriaan Jacobs and Stijn Volckaert, *DistriNet, KU Leuven*

Plaintext Recovery Against Post-Filtering Access Control 253
Zachary Espiritu, *MongoDB Research;* David Cash, *University of Chicago*

PICASSO: Scaling CHERI Use-After-Free Protection to Millions of Allocations using Colored Capabilities 259
Merve Gülmez, *Ericsson Security Research;* Ruben Sturm, *DistriNet, KU Leuven;* Hossam ElAtali, *University of Waterloo;* Håkan Englund, *Ericsson Security Research;* Jonathan Woodruff, *University of Cambridge;* N. Asokan, *University of Waterloo and KTH Royal Institute of Technology;* Thomas Nyman, *Ericsson Product Security*

Trust Nothing: RTOS Security without Run-Time Software TCB 265
Eric Ackermann and Sven Bugiel, *CISPA Helmholtz Center for Information Security*

Human Aspects 2

Cross-National Information Attacks: A Two-Decade Analysis of Troll Behavior in Korea 271
Jaehong Kim, *Korea Advanced Institute of Science and Technology* (*KAIST*) *and Max Planck Institute for Security and Privacy* (*MPI-SP*)*;* Hyeonseung Kim, *Korea Advanced Institute of Science and Technology* (*KAIST*)*;* Jiseon Kim, *Korea Advanced Institute of Science and Technology* (*KAIST*) *and Max Planck Institute for Security and Privacy* (*MPI-SP*)*;* Alice Oh, *Korea Advanced Institute of Science and Technology* (*KAIST*)*;* Thorsten Holz, *Max Planck Institute for Security and Privacy* (*MPI-SP*)*;* Wonjae Lee, *Korea Advanced Institute of Science and Technology* (*KAIST*)*;* Meeyoung Cha, *Max Planck Institute for Security and Privacy* (*MPI-SP*) *and Korea Advanced Institute of Science and Technology* (*KAIST*)

Characterizing Security and Privacy Teaching Standards for Schools in the United States 273
Katherine Limes, *Colorado School of Mines;* Nathan Malkin, *New Jersey Institute of Technology;* Kelsey R. Fulton, *Colorado School of Mines*

Software Security 3

BugAuditor: Detecting Bugs via Inconsistent Defensive Code Auditing 275
Miaoqian Lin, *The University of Hong Kong;* Kai Chen, *Institute of Information Engineering, Chinese Academy of Sciences;* Hao Chen, *The University of Hong Kong*

FirmReBugger: A Benchmark Framework for Monolithic Firmware Fuzzers 277
Mathew Duong, Michael Chesser, and Guy Farrelly, *University of Adelaide;* Surya Nepal, *Data61 CSIRO;* Damith C. Ranasinghe, *University of Adelaide*

RecStruct: Recovering Nested Struct Types from Stripped Binaries via Stack-Driven Unification 281
Yuxin Chen, Zhiyang Fang, Shiyi Wu, Yixin Xu, and Yuhang Wang, *Sichuan University;* Xiaokang Yin, *Information Engineering University;* Junfeng Wang, *Sichuan University*

CombiSan: Unifying Software Sanitizers for Comprehensive Fuzzing 283
Matteo Marini, *Sapienza University of Rome;* Floris Gorter, *Vrije Universiteit Amsterdam;* Daniele Cono D'Elia, *Sapienza University of Rome;* Cristiano Giuffrida, *Vrije Universiteit Amsterdam*

Applications of Cryptography 4

InstantOMR: Oblivious Message Retrieval with Low Latency and Optimal Parallelizability 285
Haofei Liang, *Shanghai Jiao Tong University;* Zeyu Liu, *Yale University;* Eran Tromer, *Boston University;* Xiang Xie, *Primus Labs;* Yu Yu, *Shanghai Jiao Tong University*

Message Injection Attacks Against Signal 289
Kien Tuong Truong, *ETH Zurich;* Noemi Terzo, *Max Planck Institute for Security and Privacy;* Kenneth G. Paterson, *ETH Zurich*

Cryptanalysis of LDPC-Based Pseudorandom Error-Correcting Codes 293
Tianrui Wang, *Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University;* Anyu Wang, *Institute for Advanced Study and State Key Laboratory of Cryptography and Digital Economy Security, Tsinghua University; Zhongguancun Laboratory;* Tianshuo Cong, *School of Cryptologic Science and Engineering and Shandong Key Laboratory of Artificial Intelligence Security, Shandong University;* Delong Ran, *Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University;* Jinyuan Liu, *Institute for Advanced Study, Tsinghua University;* Xiaoyun Wang, *Institute for Advanced Study and State Key Laboratory of Cryptography and Digital Economy Security, Tsinghua University; Zhongguancun Laboratory; National Financial Cryptography Research Center; Shandong Institute of Blockchain*

Security of ML 4

On Improving Robustness of Deepfake Image Detectors 297
Abu Taib Mohammed Shahjahan, Mohammad Mannan, Abdessamad Ben Hamza, and Amr Youssef, *Concordia University*

Adversarial Patch EXterminator: Zero-Shot and Patch-Agnostic Defense Framework Against Adversarial Patch Attacks. 301
Jiayimei Wang, *City University of Hong Kong;* Tao Ni, *King Abdullah University of Science and Technology;* Guowen Xu, *University of Electronic Science and Technology of China;* Qingchuan Zhao and Cong Wang, *City University of Hong Kong*

Friday, August 14

Applications of Cryptography 5

Your Keywords Know Each Other: Breaking SSE with <1% Leaked Documents 305
Mingyu Bian, Jiabei Wang, Dandan Xu, Guangyu Huang, and Yongbin Zhou, *School of Cyber Science and Engineering, Nanjing University of Science and Technology*

Sonic: Concurrent Oblivious RAM & Data Structures for Low-Latency and High-Throughput 309
Nihal Talur and Ioannis Demertzis, *UC Santa Cruz*

DDR-SSE: Duplicated Retrieval of Documents for System-wide Secure Searchable Symmetric Encryption 313
Zichen Gui, *University of Georgia;* Simon-Philipp Merz and Kenneth G. Paterson, *ETH Zürich;* Sikhar Patranabis, *IBM Research India*

mmCipher: Batching Post-Quantum Public Key Encryption Made Bandwidth-Optimal 317
Hongxiao Wang, *The University of Hong Kong;* Ron Steinfeld, *Monash University;* Markku-Juhani O. Saarinen, *Information Security Laboratory, Tampere University;* Muhammed F. Esgin, *Monash University;* Siu-Ming Yiu, *The University of Hong Kong*

ZipPIR: High-throughput Single-server PIR without Client-side Storage. 321
Rasoul Akhavan Mahdavi, Abdulrahman Diaa, and Florian Kerschbaum, *University of Waterloo*

Shadowfax: Hybrid Security and Deniability for AKEMs. 323
Phillip Gajland, *IBM Research Europe – Zurich;* Vincent Hwang, *MPI-SP, Radboud University;* Jonas Janneck, *Ruhr University Bochum*

Security of ML 5

HijackKV: New Threat in Position-Independent KV Cache Reuse. 327
Yichi Zhang and Zhiqi Wang, *The Pennsylvania State University;* Huan Zhang, *University of Illinois Urbana-Champaign;* Yuchen Yang, *The Pennsylvania State University*

Low-Cost Hard-Label Adversarial Attack with Theoretical Foundations 333
Jun Liu, *University of Macau; National Institute of Informatics;* Leo Yu Zhang, *Griffith University;* Fengpeng Li, *University of Macau;* Isao Echizen, *University of Tokyo; National Institute of Informatics;* Jiantao Zhou, *University of Macau*

Security and Privacy for Web, Mobile, and Emerging Technologies 3

BUIzz: Finding Policy Enforcement Bugs via Interaction Simulation on the Browser User Interface 337
Mingi Jung, Donggyu Kim, Mijung Kim, and Seongil Wi, *UNIST*

The State of Passkeys: Studying the Adoption and Security of Passkeys on the Web 341
Louis Jannett, *Ruhr University Bochum;* Andreas Mayer and Maximilian Westers, *Heilbronn University of Applied Sciences;* Vladislav Mladenov, *Ruhr University Bochum;* Christian Mainka, *University of Wuppertal;* Jörg Schwenk, *Ruhr University Bochum*

Software Security 4

Meerkat: Pushing the Practical Limits of Dynamic Bisection with PoC Mutation 345
Joseph Bursey, Christoph Sendner, and Ardalan Amiri Sani, *UC Irvine;* Zhiyun Qian, *UC Riverside*

BenchChecker: Assessing the Credibility of Bug-Fixing Benchmarks for LLMs 349
Di Wu and Xu He, *George Mason University;* Shu Wang, *Palo Alto Networks, Inc.;* Kun Sun, *George Mason University*

VeCT: Secure and Efficient Constant-Time Code Rewriting with Vector Extensions 351
Qisheng Jiang and Danfeng Zhang, *Duke University*

CULPA: Universal Detection of Memory-Safety Bugs in Unsafe Rust Through the Lens of Safety Requirements 353
Hung-Mao Chen and Bo Lu, *George Mason University;* Xu He, *Visa Inc.;* Xiaokuan Zhang and Kun Sun, *George Mason University*

Wormholes in the File System: Understanding the Misunderstanding of Symlinks 355
Yongheng Liu, Lei Zhang, Yuhang Zhao, and Yuzhou He, *Fudan University*

Hardware Security 4

PROBE+DETECT+MITIGATE (PDM): Enabling Cloud Tenants to Self-Defend against Microarchitectural Attacks 357
Arash Daneshmand, *University of British Columbia Okanagan and Concordia University;* Hugo Kermabon-Bobinnec, *Concordia University;* Lingyu Wang, *University of British Columbia Okanagan and Concordia University;* Makan Pourzandi, *Ericsson Security Research, Ericsson Canada;* Suryadipta Majumdar, *Concordia University;* Yosr Jarraya, *Ericsson Security Research, Ericsson Canada*

Charting the Cache Side-Channel Frontier: A Systematic Study of Eviction Set Construction on Apple Silicon 363
Han Wang, Yakun Wu, Yusi Feng, and Yinqian Zhang, *Southern University of Science and Technology*

microSCALE: Static Analysis for Microarchitectural Side-channel Leakage Evaluation 367
Akshay Kumar E, Pranav Krishna N, Annapurna Valiveti, Pallavi Borkar, Aditi Roy, and Chester Rebeiro, *IIT Madras*

Spectre on RISC-V Silicon: Attacks and Defenses on Commercial Out-of-Order Processors 369
Lukas Gerlach, *CISPA Helmholtz Center for Information Security;* Marton Bognar, *DistriNet, KU Leuven;* Daniel Weber and Michael Schwarz, *CISPA Helmholtz Center for Information Security;* Jo Van Bulck, *DistriNet, KU Leuven*

UncoreBleed: AEX-Free, High-Resolution, and Low-Noise Side-Channel Attacks on SGX Enclaved Execution 373
Decheng Chen, *South China University of Technology;* Zhi Zhang, *The University of Western Australia;* Zhenkai Zhang, *Clemson University;* Xin Zhang, *Shandong University;* Yansong Gao, *Southeast University;* Yi Zou, *South China University of Technology*

Exploiting Hidden Resource Contention in Selective Speculation Defenses 377
Xiaoyu Cheng, *Southeast University;* Fei Tong, *Southeast University; Purple Mountain Laboratories; Jiangsu Province Engineering Research Center of Security for Ubiquitous Network;* Zhenyu Lei, Fang Jiang, Zhe Zhou, and Guang Cheng, *Southeast University;* Trevor E. Carlson, *National University of Singapore* (*NUS*)

Human Aspects 3

What Adults Will (and Won't) Do to Prove Their Age: Empirical Evidence from a Deceptive Web Experiment 381
Yanzi Lin and Cheng Zhang, *Carnegie Mellon University;* Madelyne Xiao, *Princeton University;* Lorrie Faith Cranor and Sarah Scheffler, *Carnegie Mellon University*

Hardware Security 5

XCFI: Comprehensive Control-Flow Integrity for Arm TrustZone-M 385
Yunju Gu, Jaeyeol Park, and Donghyun Kwon, *Pusan National University*

StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine **387**
Ruiyi Zhang, Tristan Hornetz, Daniel Weber, Fabian Thomas, and Michael Schwarz, *CISPA Helmholtz Center for Information Security*

M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M **389**
Cristiano Rodrigues, *Centro ALGORITMI, Universidade do Minho;* Marton Bognar, *DistriNet, KU Leuven;* Sandro Pinto, *Centro ALGORITMI, Universidade do Minho;* Jo Van Bulck, *DistriNet, KU Leuven*

Memclave: Secure In-Memory Enclave for Untrusted Hosts **393**
Amit Choudhari, *CISPA Helmholtz Center for Information Security;* Fabian van Rissenbeck, *Technische Universität Dortmund;* Christian Rossow, *CISPA Helmholtz Center for Information Security*

Applications of Cryptography 6

Rarus: A Succinct and Efficient Range Proof for Polynomial-based Vector Commitment **397**
Xinyang Yang, Wenjie Qu, Yanpei Guo, and Jiaheng Zhang, *National University of Singapore*

Prezta: Provable Remote Execution of Zero-Trust Authorization using SNARKs **399**
Zhongjing Wei, Osaid Muhammad Ameer, Yupeng Zhang, and Nikita Borisov, *University of Illinois Urbana-Champaign*

Efficiently Provable Approximations for Non-Polynomial Functions **403**
Sriram Sridhar, *University of California, Berkeley;* Shravan Srinivasan, *Lagrange Labs;* Dimitrios Papadopoulos, *Hong Kong University of Science and Technology;* Charalampos Papamanthou, *Lagrange Labs and Yale University*

Shred-to-Shine Metamorphosis of (Distributed) Polynomial Commitments **407**
Weihan Li, *School of Cyber Science and Technology, Beihang University; Ant Group;* Zongyang Zhang, *School of Cyber Science and Technology, Beihang University;* Sherman S. M. Chow, *The Chinese University of Hong Kong;* Yanpei Guo, *National University of Singapore;* Boyuan Gao, *School of Cyber Science and Technology, Beihang University;* Xuyang Song, *Anoma;* Yi Deng, *School of Cryptology, Xidian University;* Jianwei Liu, *School of Cyber Science and Technology, Beihang University*

Security of ML 6

Quantifying Large Language Model Attacks Through the Lens of Model Cognition **411**
Xiuming Liu, Chaoxiang He, Xuanran Yu, Jichen Chai, Feiyue Xu, Sheng Hang, and Hanqing Hu, *Shanghai Jiao Tong University;* Bin Benjamin Zhu, *Microsoft Corporation;* Hongsheng Hu, Shi-Feng Sun, Dawu Gu, and Shuo Wang, *Shanghai Jiao Tong University*

Certified in Theory, Broken in Practice: Assumption Gaps in Cryptographic Model Certification **415**
Carter Luck, *University of Massachusetts Amherst;* Olive Franzese-McLaughlin, *Vector Institute and University of Toronto;* Elisaweta Masserova, *Carnegie Mellon University;* Akira Takahashi and Antigoni Polychroniadou, *J.P. Morgan AI Research and AlgoCRYPT CoE;* Nicolas Papernot, *Vector Institute and University of Toronto*

MATE: Policy-Aware Security Auditing for Mobile Agents via Synthesis-Driven Trajectory Learning **419**
Changyue Jiang, *Fudan University and Shanghai Innovation Institute;* Jiayi Wang, Xin Wen, Jiarun Dai, and Geng Hong, *Fudan University;* Xudong Pan, *Fudan University and Shanghai Innovation Institute*

Network Security 4

When Address Learning Goes Wrong: Inducing Forwarding Loops and DoS Amplification in SDN **423**
Dezhang Kong, *Zhejiang University;* Yilun Zhang, *Purdue University;* Zekun Xie, *Xi'an University of Posts and Telecommunications;* Ningpeng Zheng, *Yancheng Institute of Technology;* Shi Lin, *Zhejiang Gongshang University;* Zhenhua Xu, *Zhejiang University;* Minghao Li, *Chongqing University;* Zhebo Wang, Xiang Chen, and Changting Lin, *Zhejiang University;* Dong Zhang, *Fuzhou University;* Xuan Liu, *Yangzhou University;* Chunming Wu and Meng Han, *Zhejiang University*

DNS Cache Poisoning Like it's 2006 . . . . . **431**
Omer Ben-Simhon and Amit Klein, *Hebrew University of Jerusalem*

Side-Channel Attacks on Open vSwitch **435**
Daewoo Kim and Sihang Liu, *University of Waterloo*

Applications of Cryptography 7

Concretely Efficient Blind Signatures Based on VOLE-in-the-Head Proofs and the MAYO Trapdoor 437
Carsten Baum and Marvin Beckmann, *Denmark Technical University;* Ward Beullens, *IBM, Zürich;* Shibam Mukherjee, *Graz University of Technology and Know Center;* Christian Rechberger, *Graz University of Technology and TACEO*

Heli: Heavy-Light Private Aggregation 439
Ryan Lehmkuhl and Henry Corrigan-Gibbs, *MIT;* Emma Dauterman, *Stanford;* David J. Wu, *University of Texas at Austin*

Quorus: Efficient, Scalable Threshold ML-DSA Signatures from MPC 441
Alexander Bienstock, Leo de Castro, Daniel Escudero, Antigoni Polychroniadou, and Akira Takahashi, *J.P. Morgan AlgoCRYPT CoE and J.P. Morgan AI Research*

Anonymous Tokens with Designated-Reader Metadata Bit 443
Aisha Tu, *Wuhan University;* Meng Jia, *The Hong Kong Polytechnic University;* Kun He, Jing Chen, and Ruiying Du, *Wuhan University*

Privacy of ML 2

VOID: Defeating Unauthorized Mimicry in Latent Diffusion Models 445
Chunlin Qiu, Ang Li, Tianxiao Huang, Ruilin Gan, Yunjie Ge, and Shenyi Zhang, *Wuhan University;* Huayi Duan, *The Hong Kong University of Science and Technology* (*Guangzhou*)*;* Lingchen Zhao, *Wuhan University;* Chao Shen, *Xi'an Jiaotong University;* Qian Wang, *Wuhan University*

From Easy to Hard++: Promoting Differentially Private Image Synthesis Through Spatial-Frequency Curriculum 449
Chen Gong and Kecen Li, *University of Virginia;* Zinan Lin, *Microsoft Research;* Tianhao Wang, *University of Virginia*

CompLeak: Deep Learning Model Compression Exacerbates Privacy Leakage 451
Na Li, *School of Cyber Science and Engineering, Nanjing University of Science and Technology, China;* Yansong Gao, *School of Cyber Science and Engineering, Southeast University, China;* Hongsheng Hu, *School of Computer Science, Shanghai Jiao Tong University, China;* Boyu Kuang, *School of Cyber Science and Engineering, Nanjing University of Science and Technology, China;* Anmin Fu, *School of Cyber Science and Engineering, Nanjing University of Science and Technology, China; and School of Computer Science and Engineering, Nanjing University of Science and Technology, China*

Human Aspects 4

From Assistance to Autonomy: An Empirical Study of AI Use in a Live Capture-the-Flag (CTF) Competition. . . . 455
Tingxuan Tang, Nicolas Janis, and Kalyn Asher Montague, *William & Mary;* Kevin Eykholt, Dhilung Kirat, Youngja Park, and Jiyong Jang, *IBM Research;* Adwait Nadkarni and Yue Xiao, *William & Mary*

Chameleon Channels: Measuring YouTube Accounts Repurposed for Deception and Profit 459
Alejandro Cuevas, *Carnegie Mellon University;* Manoel Horta Ribeiro, *Princeton University;* Nicolas Christin, *Carnegie Mellon University*

Not All Those Who Share Are Lost: Analyzing 25 Years of Cybersecurity Artifact Sharing Practices Through Automated Discovery 463
Daan Vansteenhuyse, Arthur Bols, and Lieven Desmet, *DistriNet, KU Leuven;* Victor Le Pochat, *Independent;* Jo Van Bulck and Marton Bognar, *DistriNet, KU Leuven*

System Security 3

PATCHWEAVER: Risk-Bounded Autonomous Vulnerability Remediation Under Change-Management Policies 465
Rui Li and Shuang Cao, *Dawn Computing*

Trustworthy and Confidential SBOM Exchange. 469
Eman Abu Ishgair and Chinenye Okafor, *Purdue University;* Marcela S. Melara, *Intel Corporation;* Santiago Torres-Arias, *Purdue University*

Concurrency Fuzzing of the Linux Kernel with eBPF 473
Jiacheng Xu, *Zhejiang University;* Dylan Wolff, Xing Yi Han, Jialin Li, and Abhik Roychoudhury, *National University of Singapore*

Security and Privacy for Web, Mobile, and Emerging Technologies 4

XGuardian: Towards Generalized, Explainable and More Effective Server-side Anti-cheat in First-Person Shooter Games 477
Jiayi Zhang, Chenxin Sun, and Chenxiong Qian, *The University of Hong Kong*

Send and Pretend: Exploiting Transcript Consistency Issues in End-to-End Encrypted Group Chats 489
Gabriel K. Gegenhuber, *Interdisciplinary Transformation University* (*IT:U*)*;* Moritz Grefner, *University of Vienna;* Maximilian Günther, *SBA Research;* Matthäus Wininger, *University of Vienna;* David Schmidt, *University of Vienna, UniVie Doctoral School Computer Science, and CDL AsTra;* Aljosha Judmayer, *University of Vienna*

Relay and Betray: Exploiting Client-Side Authority in Multi-User Mixed Reality 493
Mutahar Ali and Habiba Farrukh, *University of California, Irvine*



USENIX Security '26 Artifact Appendix: Orbit: Optimizing Rescale and Bootstrap Placement with Integer Linear Programming Techniques for Secure Inference

Zikai Zhou*† William Seo*‡ Edward Chen‡ Alex Ozdemir§ Fraser Brown‡ Wenting Zheng‡
†Tsinghua University ‡Carnegie Mellon University §Max Planck Institute for Security and Privacy
zhouzk22@mails.tsinghua.edu.cn, {wys, ejchen, fraserb, wenting}@cmu.edu, alex.ozdemir@mpi-sp.org

A Artifact Appendix

A.1 Abstract

This artifact contains Orbit, an FHE compiler that jointly optimizes bootstrap and rescale placement for CKKS secure-inference workloads through a level-scale-aware integer linear programming (ILP) formulation. It includes the Orbit implementation (about 3,500 lines of Python and 2,500 of Go), MLIR inputs for five CNNs (ResNet-20, SqueezeNet, AlexNet, MobileNet, VGG16) in SiLU and ReLU variants, profiled Lattigo cost models, and scripts that reproduce the paper's experiments. The artifact packages Orbit itself; the baseline placement algorithms (DaCapo [3], Orion [4], ReSBM [6]) are not included, so it reproduces Orbit's own results and compares them against the baseline numbers reported in the paper. Reviewers can compile every benchmark and check Orbit's compilation times (Tables 6–7), estimate or execute the compiled programs to check Orbit's latencies and operation counts (Figures 5–6, Table 8), and verify accuracy within 0.3% of plaintext PyTorch (Table 9). All experiments target CKKS in Lattigo on a standard Linux machine.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. All experiments are purely computational, run in user space, and use only synthetic inputs and public CIFAR-10 samples.

A.2.2 How to access

An archived version matching the evaluated paper is on Zenodo (`https://zenodo.org/records/20278435`); ongoing development is on GitHub (`https://github.com/cmu-cryptosystems/Orbit`).

*Equal contribution.

A.2.3 Hardware dependencies

Orbit needs only a commodity CPU and under 5 GB of RAM to compile. To reproduce the paper we recommend a high core count (QBP solving is parallelized) and, for executing the larger benchmarks, up to 256 GB of RAM; the paper compiled on AWS `m8i.8xlarge` (32 cores) and executed on `m7g.16xlarge` (64 cores, 256 GB). Allow about 20 GB of disk for compilation and one benchmark's execution data; the plaintext constants for *all* benchmarks total order 50 GB.

A.2.4 Software dependencies

Orbit was tested with Python 3 (the Docker image pins Python 3.11 on Debian, with Go 1.24 for the backend). Python dependencies (`gurobipy`, `joblib`, `matplotlib`, `more-itertools`, `networkx`, `numpy`) are in `requirements.txt` and installable with `pip`. The `gurobipy` package ships a size-limited license sufficient for every standard (partitioned) benchmark; only the `-nopart` and `-nocomp` ablations exceed its cap and need a full (e.g., free academic) Gurobi license. The DaCapo frontend (generates plaintext inputs) and Lattigo backend (executes compiled MLIR) are needed only for execution; **reviewers checking only estimated results need neither.** See `frontend/README.md` and `backend/README.md`, or build both inside the container with `scripts/setup_dependencies.sh`.

A.2.5 Benchmarks

Five CNNs used by prior FHE compilers—ResNet-20, SqueezeNet, AlexNet, MobileNet, VGG16—each in a SiLU variant (single-Chebyshev SiLU, average-pooling) and a ReLU variant (three-Chebyshev ReLU, max-pooling; pooling only in AlexNet, SqueezeNet, VGG16). They are MLIR files in `mlirs_input/`, lowered by DaCapo into Orbit's FHE Computation IR (the paper's shared input representation). Accuracy is validated against plaintext PyTorch on CIFAR-10.

A.3 Set-up

A.3.1 Installation

Install the Python dependencies:

```
git clone \
  https://github.com/cmu-cryptosystems/Orbit
cd Orbit && pip install -r requirements.txt
```

To execute (not only estimate) programs, also install the frontend and backend (their READMEs), or use the Docker image below—which needs no manual Python or Gurobi setup.

A.3.2 Basic Test

Run `pytest`, then compile one benchmark end to end:

```
python3 scripts/optimizer/orbit/run_orbit.py \
  --model ResNet --act SiLU \
  --n 64 --Lm 16 --Sw 40
```

Output lands under `mlirs_output/`: the `.txt` log reports `Orbit Compilation time` and the estimated `Final tdag latency`; the `.mlir` is the compiled program.

A.3.3 Reproducing the experiments with Docker

Build the image once and drive it with `scripts/reproduce.sh`; the image bundles the dependencies and the size-limited Gurobi license, so the cost-model experiments (E1, E3) need no further setup:

```
docker build --build-arg USER_ID=$(id -u) \
    --build-arg GROUP_ID=$(id -g) -t orbit:dev .
docker run --rm orbit:dev   # tests+smoke
```

The experiments below run `reproduce.sh` targets through this image; mounting the repository with `-v $PWD:/opt/orbit` keeps generated files (compiled MLIRs and logs) on the host. The `-nopart`/`-nocomp` CompPart ablations additionally need a full Gurobi license, added with `-v $HOME/gurobi.lic:/opt/gurobi/gurobi.lic:ro` and `-e GRB_LICENSE_FILE=/opt/gurobi/gurobi.lic`. To execute (E2, E4), first build the backend and frontend and generate inputs once, then use the `execute` target:

```
D="docker run --rm -e HOME=/tmp \
   -v $PWD:/opt/orbit"
$D orbit:dev \
    ./scripts/setup_dependencies.sh backend
$D orbit:dev \
    ./scripts/setup_dependencies.sh frontend
$D -w /opt/orbit/frontend/dacapo orbit:dev \
    bash -c 'source .venv/bin/activate && \
    ../dacapo_patch/gen_all_mlirs.sh && \
    python3 examples/tests/gen_input_data.py 10'
$D orbit:dev ./scripts/reproduce.sh execute \
    --model ResNet --act SiLU \
    --n 16 --Lm 16 --Sw 40 --run 0
```

Decrypted output lands in `mlirs_execute/`, comparable to the PyTorch reference from the data-generation step (E4).

A.4 Evaluation workflow

A.4.1 Major Claims

The baselines (DaCapo, Orion, ReSBM) are **not** part of this artifact; each claim is reproduced for Orbit, and the paper computes the comparisons against the reported baseline numbers.

(C1): Orbit compiles every benchmark and configuration in under 6 minutes; the reproduced times are the Orbit columns of Tables 6–7. Verified by E1.

(C2): Orbit's joint bootstrap-and-rescale ILP lowers execution cost: its compiled programs use up to 5.4× fewer rescales and 31–34% fewer bootstraps than Orion (Figures 5–6, Table 8), yielding the paper's speedups of 7–19% over DaCapo, 2–73% over Orion, and up to 52% over ReSBM. Verified by E2.

(C3): Orbit's DAG-reduction techniques (compression, partitioning, QBP reuse, bypass handling) cut ILP solving time while keeping end-to-end latency within 1% (Tables 3–5). Verified by E3.

(C4): Orbit stays within 0.3% of plaintext PyTorch accuracy across all benchmarks (Table 9). Verified by E4.

A.4.2 Experiments

Each suite below is shown as its native script `auto_scripts/<name>.sh`; equivalently, run it through the Docker image as `reproduce.sh <t>`—short for `docker run -rm -v $PWD:/opt/orbit orbit:dev ./scripts/reproduce.sh <t>`—whose targets mirror the script names (e.g., `compile_orbit_eval_base.sh` ≡ `compile-base`). Execution (E2, E4) additionally needs `-e HOME=/tmp` and the one-time backend/frontend build from Set-up. Below, the instructions use the commands for compilation and execution in Docker.

(E1): Compilation time [15 human-minutes + several compute-hours]. Compile the four configuration suites (base n=64k, plus 16k, L_{bts}=12, and S_{min}=51):

```
reproduce.sh compile-base
reproduce.sh compile-16k
reproduce.sh compile-lm12
reproduce.sh compile-sw51
```

Each `.txt` log under `mlirs_output/` reports `Orbit Compilation time`: on the recommended multi-core machine every benchmark compiles in under 6 minutes, matching the Orbit columns of Tables 6–7. Compilation parallelism follows `ORBIT_THREADS` (default 32; set it to the machine's core count).

(E2): Runtime performance [10 human-minutes + ∼1-hour setup + minutes–hours per benchmark]. Install the frontend and backend (or use Docker) to execute, or skip for estimates only. Compile as in E1, then run each program through the `execute` target (the command shown in Set-up) on Lattigo; `compile-sim-vari` selects the variable-bootstrapping cost model for the ReSBM comparison. Compare Orbit's latencies and the rescale/bootstrap counts printed in each `.txt` log against the Orbit data in Figures 5–6 and Table 8. Building the frontend/backend and generating inputs is a one-time ∼1-hour step; plaintext evaluation then takes seconds to minutes (∼10 s for AlexNet-SiLU at 16k) and encrypted evaluation minutes to hours (∼25 min for AlexNet-SiLU at 16k; the 64k benchmarks run for hours and need ∼256 GB RAM).

(E3): DAG-reduction microbenchmarks [10 human-minutes + 1–2 compute-hours]. Run the ablation suites; `-nocomp`, `-nopart`, `-nobypass`, and `-qbp` on `run_orbit.py` toggle the individual techniques:

```
reproduce.sh compile-micro-comppart
reproduce.sh compile-micro-bypass
reproduce.sh compile-micro-reqbp
```

Compare compilation times, partition counts, and latencies against Table 3 (compression/partitioning), Table 4 (QBP reuse), and Table 5 (bypass); end-to-end latency stays within 1%. Each ablation compiles in seconds to ∼30 minutes. The `-nopart`/`-nocomp` variants in `compile-micro-comppart` are the slowest and exceed the bundled Gurobi license [5], so they require a full license (mounted as in Set-up); the other ablations run on the bundled license.

(E4): End-to-end accuracy [10 human-minutes + ∼1-hour setup + hours per benchmark]. Install the frontend and backend so CIFAR-10 inputs and PyTorch references are generated into `input_data/` (a one-time ∼1-hour build and data-generation step). Execute the compiled benchmarks at $n = 2^{17}$ on encrypted inputs and compare with the PyTorch outputs (Table 9): accuracy is within 0.3%, most models matching exactly (max difference order 10^{-4}). [1]

A.5 Notes on Reusability

Orbit is a modular pass over a backend-agnostic CKKS IR, so it retargets to other FHE libraries (e.g., SEAL, OpenFHE) via a new backend lowering and integrates into frameworks such as HEIR [1] and Rotom [2] through its MLIR interface. Compile arbitrary MLIR with `optimizer.py`; the core components—the ILP formulation (`ilp_core.py`), SISO partitioning and bypass handling (`siso_partition.py`), iterative solving and merging (`iterative_partition.py`), QBP management (`qbp_manager.py`), and compression (`auto_compression.py`)—are organized for extension, and new backend/hardware cost models drop into `cost_models/` as profiled JSON.

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/usenixsec2026/.

References

[1] Asra Ali, Jaeho Choi, Bryant Gipson, Shruthi Gorantala, Jeremy Kun, Wouter Legiest, Lawrence Lim, Alexander Viand, Meron Zerihun Demissie, and Hongren Zheng. Heir: A universal compiler for homomorphic encryption. *arXiv preprint arXiv:2508.11095*, 2025.

[2] Edward Chen, Fraser Brown, and Wenting Zheng. Bridging usability and performance: A tensor compiler for autovectorizing homomorphic encryption. *Cryptology ePrint Archive*, 2025.

[3] Seonyoung Cheon, Yongwoo Lee, Dongkwan Kim, Ju Min Lee, Sunchul Jung, Taekyung Kim, Dongyoon Lee, and Hanjun Kim. {DaCapo}: Automatic bootstrapping management for efficient fully homomorphic encryption. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 6993–7010, 2024.

[4] Austin Ebel, Karthik Garimella, and Brandon Reagen. Orion: A fully homomorphic encryption framework for deep learning. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, pages 734–749, 2025.

[5] Gurobi Optimization, LLC. Gurobi Optimizer Reference Manual, 2026.

[6] Yan Liu, Jianxin Lai, Long Li, Tianxiang Sui, Linjie Xiao, Peng Yuan, Xiaojing Zhang, Qing Zhu, Wenguang Chen, and Jingling Xue. Resbm: Region-based scale and minimal-level bootstrapping management for fhe via min-cut. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*, pages 924–939, 2025.

[1] The accuracy for $n = 2^{15}$ varies from PyTorch execution due to bugs in Dacapo frontend. As stated in the paper, this configuration is not intended for deployment.



USENIX Security '26 Artifact Appendix: Libra: Pattern-Scheduling Co-Optimization for Cross-Scheme FHE Code Generation over GPGPU

Song Bian
Beihang University

Yintai Sun
Beihang University

Zian Zhao
Beihang University

Haowen Pan
Beihang University

Mingzhe Zhang*
Unaffiliated

Zhenyu Guan*
Beihang University

A Artifact Appendix

A.1 Abstract

Libra is an end-to-end fully homomorphic encryption (FHE) compiler for GPGPUs that transforms high-level C programs into efficient GPU FHE implementations. Libra automates efficient code generation by co-optimizing cross-scheme computational patterns with hardware-aware scheduling strategies. This artifact includes: (i) the Libra compiler source code, (ii) the underlying FHE libraries (based on FlyHE), (iii) comprehensive benchmarks (microbenchmarks and end-to-end applications), and (iv) scripts that automate compilation, execution, result collection, and figure/table generation to reproduce the paper's key results.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact raises no security, privacy, or ethical concerns. It performs standard compilation and benchmarking on the included workloads and does not require disabling security mechanisms or handling sensitive data. All evaluations use only synthetic or publicly available benchmark datasets. The workflow is self-contained and does not permanently modify the host system configuration. The only external interaction is optionally pulling a public Docker image.

A.2.2 How to access

The artifact is publicly available through the following resources:
- **Source Repository (GitHub):** https://github.com/sunnchioo/Libra
- **Docker Image:** https://hub.docker.com/r/suen0/libra_ae
- **Archived Snapshot (Zenodo):** https://zenodo.org/records/17962002

*Mingzhe Zhang and Zhenyu Guan are corresponding authors.

A.2.3 Hardware dependencies

To reproduce the performance results reported in the paper, we recommend the same GPU platform used in the evaluation:
- **GPU:** NVIDIA A100 GPGPU ($\geq$ 40 GB).
- **CPU Architecture:** AMD64.
- **Host Memory:** $\geq$ 80 GB.

Other NVIDIA GPUs may run parts of the pipeline but may not reproduce performance numbers comparable to the paper.

A.2.4 Software dependencies

The artifact has been tested on Ubuntu 22.04. We provide a Docker image to ensure environment consistency.
Docker (Pre-installed): The image includes all toolchains and heavily-compiled dependencies (e.g., Polygeist, LLVM, and MLIR) pre-installed.
Manual Build Requirements:
- CMake $\geq$ 3.31.1; Ninja $\geq$ 1.10.1
- GCC/G++ $\geq$ 13.1.0; Clang/Clang++ $\geq$ 22.0.0
- CUDA Toolkit $\geq$ 12.4
- LLVM & MLIR $\geq$ 22.0
- NTL $\geq$ 11.5.1; GMP $\geq$ 6.2.1
- Python $\geq$ 3.10 (matplotlib $\geq$ 3.10.6, pandas $\geq$ 2.3.3)

A.2.5 Benchmarks

All benchmarks used in the paper are included under `Libra_full_bench`, consisting of:
- **Microbenchmarks:** Individual FHE operators.
- **Applications:** End-to-end workloads.

A.3 Set-up

A.3.1 Installation

We provide two options for installing Libra.

Option A: Docker (Recommended) We recommend using Docker to avoid dependency and toolchain mismatches.
1. Pull the Docker image:

```
docker pull suen0/libra_ae
```

2. Launch the container with GPU access:

```
sudo docker run --gpus all -it \
  --name libra_compiler \
  --memory=80g --memory-swap=84g \
  suen0/libra_ae:v1.0 /bin/bash
```

3. The source code, data, and required dependencies for the artifact are available in `Security_Artifact/Libra`.

Option B: Build from Source If not using Docker, install all dependencies listed above and build the toolchain components from the repository root (`Security_Artifact/Libra`) in the following order.

1. Build Polygeist:

```
cd Libra/Tool/Polygeist
mkdir -p build && cd build
cmake -G Ninja ../llvm-project/llvm \
  -DLLVM_ENABLE_PROJECTS="clang;mlir" \
  -DLLVM_EXTERNAL_PROJECTS="polygeist" \
  -DLLVM_EXTERNAL_POLYGEIST_SOURCE_DIR=.. \
  -DLLVM_TARGETS_TO_BUILD="host" \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_BUILD_TYPE=Release
ninja -j32
ninja check-polygeist-opt
ninja check-cgeist
```

2. Build LLVM/MLIR/Clang:

```
cd Libra/Tool/llvm-project
mkdir -p build && cd build
cmake -G Ninja ../llvm \
  -DLLVM_ENABLE_PROJECTS="mlir;clang" \
  -DLLVM_TARGETS_TO_BUILD="host" \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_BUILD_TYPE=Release
ninja -j32
ninja check-mlir
```

3. Build Libra compiler:

```
cd Libra
mkdir -p build && cd build
cmake -G Ninja ..
ninja
```

A.3.2 Basic Test

This basic test verifies that the frontend portion of the pipeline runs and produces intermediate outputs.

```
cd Libra
./Script/libra-mlir-base.sh
./Script/libra-opt-base.sh
./Script/libra-translate-base.sh
```

Expected outcome: The script completes without errors and generates base CUDA program files under the `MLIR` subdirectory in `Libra_full_bench/`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) Microbenchmark performance: Libra generates FHE kernels that outperform or match the provided reproducible baselines on microbenchmarks. This corresponds to results shown in Figure 8.
(C2) End-to-end application performance: Libra achieves significant speedups on application workloads compared to state-of-the-art baselines. These correspond to results shown in Figures 9–10, Table 4 and the ResNet-20 results.
(C3) Performance breakdown: Libra's speedups are explained by its co-optimization of cross-scheme computational patterns and hardware-aware scheduling. This corresponds to the performance breakdown shown in Figure 11.

A.4.2 Experiments

(E1): Full evaluation pipeline (reproduces C1–C3)
[2 human-hours + 3–5 compute-hours + 40GB disk]
Execution:
1. Generate CUDA code from the benchmark C programs:

```
cd Libra
./Script/libra-mlir.sh
./Script/libra-opt.sh
./Script/libra-translate.sh
python Script/libra-trans.py
```

2. Compile the foundational FHE libraries:

```
cd Libra/HElib/FlyHE
mkdir -p build && cd build
cmake ..
make -j
```

3. Run benchmarks and baselines:

```
cd Libra
./Libra_plot/run_evaluation.sh
```

To improve transparency, these intermediate IR files are intentionally preserved, enabling users to inspect the internal compiler passes, optimization stages, and cost models.
Results Collection:
Raw results are saved to:

```
Libra_plot/output/output.txt
```

To regenerate the plots and tables reported in the paper:

```
cd Libra
python ./Libra_plot/plot_all.py
```

A.5 Notes on Reusability

The artifact is designed to be modular and reusable beyond the paper. Libra is built on MLIR with a clear backend abstraction interface. Users can inspect the intermediate IR transitions to understand the cost models, and extend dialects and passes (e.g., in `Libra/Libra/Dialect` and `Libra/Libra/Tools`) to support new FHE schemes, patterns, or targets.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: VROOM: Accelerating (Almost All) Number-Theoretic Cryptography Using Vectorization and the Residue Number System

Simon Langowski
MIT

Kaiwen He
MIT

Srinivas Devadas
MIT

A Artifact Appendix

A.1 Abstract

Modular arithmetic with a large prime modulus is a dominant computational cost in number-theoretic cryptography. Modular operations are especially challenging to parallelize efficiently on CPUs using vector instructions; standard CPU implementations rely on costly carry operations and permutation instructions to align with the multiplication datapath, negating the benefits of vectorization.

We develop vectorized algorithms for modular addition and multiplication, and present a new, constant-time modular multiplication algorithm suitable for general moduli – prime or otherwise. Our method uses a Residue Number System (RNS) representation to align the arithmetic naturally with wide vector units, and strategically eliminate extraneous instructions. Existing works either require the use of customized hardware or fail to show latency improvements.

Reducing the latency of modular arithmetic results in speedups for cryptographic applications. We accelerate RSA-4096 signatures by 4.0× (verify) and 1.3× (sign) over OpenSSL, and speed up BLS signature verifications by 4.05× over the assembly-optimized `blst` library. Results on mapping our algorithm to Nvidia GPUs demonstrate speedups on modular multiplication over Nvidia's CGBN library.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None beyond ordinary compilation and benchmark execution. The artifact performs local cryptographic arithmetic on synthetic random inputs; it does not access network services, personal data, or privileged system configuration. The code is **research-grade and not audited** for production use—do not deploy it as a security product. No destructive steps or disabled security mechanisms are required.

A.2.2 How to access

Stable reference (Zenodo concept URL): https://doi.org/10.5281/zenodo.20187973

The archive contains two compressed git repositories, covering complementary parts of the evaluation. There is no dependency between the two repos: they can be evaluated on independent machines if needed.

Prerequisite: Install git. On Amazon Linux 2023 (the tested OS), run:

```
sudo dnf install git -y
```

(A) Number-theoretic CPU artifact (Sections 6.1–6.3): modular multiplication, modular exponentiation, and RSA signing/verification, implemented as a fork of BoringSSL with VROOM integrated.

```
cd ~
git clone \
  https://github.com/kevin-he-01/vroomssl.git
cd vroomssl
git checkout 5a1dc6c765a           # AE commit
# Optional: clone OpenSSL baseline
git submodule update --init
```

Step-by-step instructions and the full paper-claim mapping are in `README.md`; the helper scripts `install_deps.sh` (dependencies) and `reproduce.sh` (all of Sections 6.1–6.3) live at the repository root.

(B) Elliptic-curve / pairing (CPU) and GPU artifact (Section 6.4 onward):

```
cd ~
git clone \
  https://github.com/SimonLangowski/VROOM.git
cd VROOM
git checkout f12e58daa
```

Step-by-step instructions: `ARTIFACT.md` (CPU) and `ARTIFACT_GPU.md` (GPU). This appendix maps paper claims to commands; the per-repository guides contain full detail, pinned tool versions, and sample output.

A.2.3 Hardware dependencies

CPU artifacts (A and B) — same host requirements:

- Linux x86_64 host with **AVX-512 IFMA** (`avx512ifma` in `/proc/cpuinfo`).
- **Paper reference CPU:** AWS **c7i.metal-24xl**, Intel Xeon Platinum **8488C** (Sapphire Rapids).
- ≥8 GiB RAM; We recommend at least 30 GB of empty disk space **per repo** to be evaluated (60 GB if evaluating both repo A and repo B on the same machine). [1]
- We can provide **c7i-metal-24xl** upon request for short periods of time.

GPU artifact; separate machine from CPU:

- NVIDIA GPU with compute capability ≥7.0.
- **Paper reference GPU:** GeForce **RTX 3090** (sm_86).
- Speedups should be visible with any recent NVIDIA GPU. We are not providing ssh GPU access at this time.

A.2.4 Software dependencies

CPU path (repo A — number-theoretic, Sections 6.1–6.3):

- OS: Tested on **Amazon Linux 2023**[2]; other Linux distributions work for builds with the right dependencies.
- Compiler: **clang 21.1.8** (the official LLVM 21.1.8 prebuilt release, installed under `$HOME/LLVM-21.1.8-Linux-X64` by `./install_deps.sh`). Paper timings depend strongly on LLVM's AVX-512 IFMA codegen.
- Build tooling: **CMake**, **Ninja**, **m4** (FLINT's `configure` aborts without it), a C/C++ toolchain.
- **GMP**, **MPFR**, and **FLINT 3.2.1** — required only for the FLINT baseline columns; VROOM itself relies solely on BoringSSL `BN_*` routines. **Google Benchmark** is vendored in `third_party/` and built by CMake (no system package needed).
- Automated install: `./install_deps.sh` installs clang 21.1.8 (LLVM prebuilt) under `$HOME/LLVM-21.1.8-Linux-X64`, FLINT 3.2.1 plus Google Benchmark from source under `$HOME/.local`, and Rust/cargo 1.86.0 via `rustup` under `$HOME/.cargo` and `$HOME/.rustup` (the last needed only for the optional Arkworks baseline of Section 6.1) — all without root. The system package manager (`sudo`) installs only the base OS build tooling (including `m4`, which FLINT's `configure` requires), the GMP/MPFR development headers, and the handful of `perl-*` modules the OpenSSL baseline needs (see below). Idempotent.
- **OpenSSL baseline** (optional — only for the OpenSSL columns of Sections 6.1–6.3): a separate fork of OpenSSL fetched on demand as a git submodule by `./reproduce.sh openssl`. OpenSSL's `Configure` and build need `perl` plus the modules `FindBin`, `IPC::Cmd`, `File::Compare`, `File::Copy`, and `Time::Piece`; OpenSSL treats these as core Perl, but a minimal Amazon Linux 2023 image splits them into separate `perl-*` packages, so `./install_deps.sh` installs them via the system package manager. The baseline also needs the **Google Benchmark** that `./install_deps.sh` builds from source under `$HOME/.local` (located via `CMAKE_PREFIX_PATH`); `./reproduce.sh openssl` then compiles OpenSSL and the harness under `baselines/openssl/bench/`.

CPU path (repo B — elliptic-curve / pairing):

- OS: **Amazon Linux 2023** (paper host); Ubuntu also works for builds.
- Compiler: **clang 21.1.0** (LLVM 21.1.0 prebuilt). Paper timings depend strongly on LLVM's IFMA codegen.
- Packages: C++ toolchain, GMP, **cmake**, **git**, **curl**.
- **Google Benchmark** built from source with clang 21; using a mismatched distro `libbenchmark` with clang 21 may cause linking errors.
- Automated install: `./scripts/setup_toolchain.sh`

Vendored / pinned third-party code (included or cloned by scripts):

- **BLST** (primary CPU baseline): modified fork in `blst/`, based on `supranational/blst` @ `8065152`.
- Optional external baselines in `baselines/` (not required for the main badge): arkworks @ `598a5fba`, zkcrypto @ `6bb9695`, zksync-crypto @ `e770ffd`, gnark-crypto @ `d1dece6` (refreshed via `baselines/clone.sh`).

GPU path (repo B): Ubuntu 20.04+ (paper host), NVIDIA driver with CUDA 12.x toolkit and `nvcc`, Python 3.10+ with `numpy`, `matplotlib`, and **`cuda-python 12.x`** (`scripts/requirements-gpu.txt` pins `cuda-python>=12.0,<13`; version 13.x breaks `from cuda import cuda, cudart, nvrtc` used by `basic_benchmarker.py`). CGBN is cloned by `./scripts/setup_gpu_deps.sh`. Kernels are JIT-compiled

[1] Each repo requires a different version of the LLVM toolchain (required to reproduce exact numbers), and a full installation of each toolchain requires ∼11 GB of disk space.

[2] AMI name: `al2023-ami-2023.12.20260611.0-kernel-6.1-x86_64`

with `nvcc -std=c++20`; if the default host GCC is too old, set `NVCC_CCBIN` or `CUDAHOSTCXX` to a C++20-capable `g++`/`clang++` (or put one earlier on `PATH`) before running GPU scripts—see `ARTIFACT_GPU.md`.

A.2.5 Benchmarks

No external datasets are required. Experiments use:

- **Precomputed RNS parameters** shipped in `src/` and `scripts/*_intrns4_params.hpp` (BLS12-381 and BN254 IntRNS4 / Matrix / MatrixNoK configurations), but can be regerenated with python scripts.
- **Reference test vectors** in `test_data/` for regression tests (see `test_data/README.md`), generated using **BLST**.

A.3 Set-up

A.3.1 Installation

Compiling Repo A. Run the following commands on an IFMA-capable Linux host:

```
cd ~/vroomssl/
./install_deps.sh
./configure.sh
```

The script `install_deps.sh` installs all the dependencies needed to build and run VROOM and all baselines for repo A specificed in section A.2.4.

The last command configures and compiles the repository, producing `./build/bssl_bench`, the binary behind all of Sections 6.1–6.3. `configure.sh` selects the `Release` build type (required for accurate timings) and the home-installed clang and FLINT, so there is nothing else to set up. It is short by design — the exact compiler and CMake flags can be found in `configure.sh` (under 20 lines).

Repo B CPU host. Run the following commands on an IFMA-capable Linux host:

```
cd ~/VROOM/
./scripts/setup_toolchain.sh
```

Here `setup_toolchain.sh` (repo B) installs OS packages, LLVM 21.1.0, builds Google Benchmark with clang++, and prints `export` lines (`CXX`, `BENCHMARK_LIBS`, `BENCHMARK_INC`) for repo B's builds.

Repo B GPU host (separate machine):

```
cd ~/VROOM/
./scripts/setup_gpu_deps.sh
# if nvcc JIT fails (old default GCC): export NVCC_CCBIN=/usr/bin/g++-12
```

This installs CGBN and Python deps from `scripts/requirements-gpu.txt` (`cuda-python>=12.0,<13`). See `ARTIFACT_GPU.md` for CUDA toolkit and host-compiler notes.

A.3.2 Basic Test

Repo A smoke test.
From the `vroomssl` root, a fast (∼1 min) subset of the modular-multiplication experiment:

```
MODMUL_QUICK=1 ./reproduce.sh modmul
```

Expected outcome: a Google Benchmark table at the five table modulus sizes (1024–4096 bits) with `BM_RNSRingMultiplication` (VROOM), `BM_BSSLModMulMontgomery` (BoringSSL/OpenSSL scalar), and the FLINT rows. The VROOM rows are several times faster than both baselines at ≥1536 bits, confirming the binary, FLINT linkage, and the IFMA path all work.

Repo B CPU smoke test. Note that we export CXX from toolchain, so you may need to re-export in subsequent shells.

```
./scripts/smoke_test.sh
```

Expected outcome: the script builds BLST, runs core unit tests (`make -C src test`, `test-bls12-381-field-mul`), builds and runs the five programs under `examples/`, and prints:

```
== smoke test OK ==
```

on success (exit code 0). Any compile or test failure aborts with a non-zero exit code.

GPU smoke test. Run the following command on a CUDA host after `setup_gpu_deps.sh`

```
./scripts/smoke_test_gpu.sh
```

Expected outcome: one CGBN and one RNS GPU microbenchmark each report `correct=True`; completes in ∼5 s.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Correctness. The RNS Montgomery + CRNS implementation correctly implements BLS12-381 / BN254 field arithmetic, extension-field / FP12 operations, elliptic-curve group law, and pairing-related code paths. Proven by experiment (E1) (unit tests, examples, smoke test).

(C2): Table 11 (batch size). On the paper CPU host, amortized F_q multiplication latency decreases with batch size 1…12 as in Table 11 (register spilling at batch 12). Proven by experiment (E2); artifact rows `BM_BatchModMul_MatrixNoK_*` in `results/bench_bls12_381_table.txt`.

(C3): Table 12 (BLS field extensions). VROOM is faster than **BLST** on F_q, F_q^2, and F_q^{12} extension operations (e.g. F_q^{12} multiply $\sim$6.1×, cyclotomic square $\sim$3.6×). Proven by experiment (E2); compare `BM_FP2_*`, `BM_FP12_*` rows in `bench_bls12_381_vs_blst.txt` to Table 12.

(C4): Table 13 (BLS pairing stack). VROOM is faster than **BLST** on G1/G2 curve ops, Miller loop ($\sim$4.4×), final exponentiation ($\sim$3.7×), full pairing ($\sim$3.9×), and BLS verification ($\sim$4.1×). Proven by experiment (E2); `BM_G1_*`, `BM_MillerLoop_*`, `BM_Pairing_RNS_BLST_Inverter_*`, and computed `BM_Pairing_ComputedVerify_*` rows.

(C5): Table 10 (pairing libraries). VROOM pairing latency is lower than arkworks, zkcrypto, zksync-crypto (Pairing CE), gnark, and **BLST** on BLS12-381 (paper: **VROOM** ≈128 μs vs. **BLST** ≈504 μs). Proven by experiment (E3): `reproduce_cpu_bench.sh` plus optional `baselines/reproduce_baselines.sh bls12`.

(C6): Table 14 (BN254 G1). VROOM G1 add / mixed-add / double on BN254 is faster than arkworks, zkcrypto, and gnark-crypto (paper: ≈85–98 ns vs. 144–315 ns). Proven by experiment (E4): `make -C src bench-ec-bn254` and/or `baselines/reproduce_baselines.sh bn254`.

(C7): GPU modular multiplication (latency sweep and Figures 5–6). On RTX 3090, VROOM GPU RNS Montgomery has lower per-multiplication latency than CGBN over a 64–4096 bit modulus sweep; RNS speedup vs. CGBN approaches $\sim$4× at large moduli (Figure 5); TPI= 2t and warp-shuffle instructions improve small-modulus latency (Figure 6). Proven by experiment (E5): `./scripts/reproduce_gpu_latency.sh` (core sweep) and `scripts/gpugraph.py` (Figures 5–6).

(C8): Section 6.1, modular multiplication. On the paper CPU host, VROOM's single modular-multiplication latency is lower than `BN_mod_mul_montgomery` (the BoringSSL/OpenSSL scalar routine), FLINT, and arkworks across a 64–4096-bit modulus sweep; for moduli ≥1500 bits the speedup is ≈4× over `BN_mod_mul_montgomery` and FLINT and ≈8× over arkworks (the modular-multiplication latency/speedup figure in Section 6.1). Proven by experiment (E6); produced by repo A.

(C9): Section 6.2, modular exponentiation (Tables 7 and 9). VROOM is faster than BoringSSL and FLINT for both public-exponent ($e = 65537$) and secret-exponent modular exponentiation across 1024–4096-bit moduli (Table 7), and faster than BoringSSL for RSA-CRT exponentiation at 2048/3072/4096-bit moduli (Table 9; e.g. ≈2.3× at RSA-2048). Proven by experiment (E7); produced by repo A.

(C10): Section 6.3, RSA signatures (Table 8). VROOM lowers RSA signing and verification latency versus BoringSSL for RSA-2048/3072/4096 (sign ≈2.3–3.2×, verify ≈3.3–3.6×). Versus OpenSSL (paper abstract: RSA-4096 *verify* 4.0×, *sign* 1.3×), the OpenSSL column is an *external* baseline produced by a separate forked-OpenSSL harness in `baselines/openssl` (run via `./reproduce.sh openssl rsa`); `bssl_bench` itself directly reproduces the VROOM and BoringSSL columns. Proven by experiment (E8); produced by repo A.

A.4.2 Experiments

(E1): Functional verification [15 human-minutes+ 3–5 compute-minutes + ∼500 MB disk]

How to: Confirm the artifact builds and passes regression tests and minimal examples.

Preparation: Complete Set-up (Installation) above on an IFMA host, or set `FALLBACK=1` on a non-IFMA host if you're just testing correctness.

Execution:
```
./scripts/smoke_test.sh
# optional extended tests:
make -C src test-pairing test-ec \
    test-bls12-381-pairing
make -C examples && \
    ./examples/01_parameter_setup
```

Results: All commands exit 0. Smoke test prints `== smoke test OK ==`. Extended tests print pass summaries with no assertion failures.

(E2): Tables 11–13: BLS12-381 CPU AVX-512 IFMA vs. BLST [60 human-minutes + ∼2 compute-minutes + ∼8 MB `results/`]

How to: Reproduce Google Benchmark rows that populate Tables 11–13 (MatrixNoK configuration on c7i.metal-24xl).

Preparation: AWS c7i with IFMA:
```
eval \
"$(./scripts/setup_toolchain.sh --print)"
```

Execution: `./scripts/reproduce_cpu_bench.sh`

Results: Use `results/bench_bls12_381_table.txt` and `bench_bls12_381_vs_blst.txt` (ratio = `blst_ns/rns_ns`; > 1 means VROOM is faster).

Table 11: amortized ns from `BM_BatchModMul_MatrixNoK_{1,2,4,6,8,10,12}`.

Table 12: `BM_ModMul_MatrixNoK`, `BM_FP2_Mul_MatrixNoK`, `BM_FP12_Mul_MatrixNoK`, `BM_FP12_CyclotomicSqr_MatrixNoK`, `BM_FP12_Inverse_RNS_BLST_MatrixNoK` vs. aligned BLST rows (speedups ≈1.3–6.1×) Python script contains mapping of VROOM to BLST benchmark names.

Table 13: `BM_G1_*`, `BM_G2_*`, `BM_MillerLoop_MatrixNoK` (≈50 μs), `BM_FP12_FinalExp_BLST_Inverter_MatrixNoK`, `BM_Pairing_RNS_BLST_Inverter_MatrixNoK` (≈128 μs), and computed

BM_Pairing_ComputedVerify_MatrixNoK (≈4× over BLST formula sum).

(E3): Table 10: BLS12-381 pairing vs. external libraries [45 human-minutes + ∼30 compute-minutes + several GiB baselines/]

How to: Compare full pairing latency across arkworks, zkcrypto, zksync-crypto, gnark-crypto, **BLST**, and VROOM.

Preparation: Rust, Go, and CPU toolchain (§ Software dependencies). Vendored trees under baselines/; pinned commits in baselines/clone.sh.

Execution: Run the following two commands:

```
# BLST
./scripts/reproduce_cpu_bench.sh
# arkworks, zkcrypto, zksync, gnark
./baselines/reproduce_baselines.sh bls12
```

Pairing times are in baselines_bls12_bench.log (Criterion / Go ns/op) and bench_bls12_381_table.txt (BM_Pairing_RNS_BLST_Inverter_MatrixNoK, BM_FullPairing in BLST JSON).

Results: Manually align pairing rows to Table 10 (no automated parser).

(E4): Table 14: BN254 G1 elliptic-curve operations versus external libraries [30 + ∼10 compute-minutes]

How to: Compare BN254 G1 add, mixed-add, and double latency.

Preparation: Same as E2 and E3, but with bn254 verison of script.

Execution: `make -C src bench-ec-bn254`
`./baselines/reproduce_baselines.sh bn254`

Results: ./src/bench_ec_bn254 JSON or stdout for BM_G1_Add_MatrixNoK, BM_G1_MixedAdd_MatrixNoK, BM_G1_Double_MatrixNoK. Compare to arkworks / zkcrypto / gnark rows in baselines_bn254_bench.log. Paper reference: VROOM ≈85–98 ns vs. baselines 144–315 ns.

(E5): GPU latency sweep and Figures 5–6 [20–30 human-minutes GPU setup + ∼8 compute-minutes + ∼50 MB results/ and scripts/data/]

How to: Run the full GPU modular-multiplication latency sweep (CGBN vs. VROOM RNS), then regenerate the supplementary GPU plots for Figures 5–6.

Preparation: RTX 3090 (or similar), CUDA 12.x toolkit, and ./scripts/setup_gpu_deps.sh (installs cuda-python>=12.0,<13). If JIT compile fails because the default host GCC is too old for C++20, set NVCC_CCBIN or CUDAHOSTCXX to a newer g++ before running (see ARTIFACT_GPU.md). For Figures 5–6, scripts/gpugraph.py reads pinned CSVs under scripts/data/ (listed in the script header); the latency sweep writes a fresh CSV under scripts/data/<timestamp>.csv and copies to results/gpu_latency.csv.

Execution: From the repo root (both steps required):

```
./scripts/reproduce_gpu_latency.sh
cd scripts
export MPLBACKEND=Agg
python3 gpugraph.py
```

Step 1 compiles many nvcc kernels (dominates wall time). Step 2 is fast if using the shipped scripts/data/*.csv inputs; re-point gpugraph.py to the newest sweep CSV if regenerating all plots from scratch.

Results: Latency sweep: results/gpu_latency.csv, results/gpu_latency.png, and results/gpu_latency_resources.txt (paper host: latency_sweep_elapsed_s=491.68 on RTX 3090, 262 sweep rows; peak RSS ∼1.8 GiB). Every CSV row should have correct=True. RNS should beat CGBN in the mid-bitrange. **Figures 5–6:** scripts/CGBNBaseline.png (Figure 5: speedup vs. CGBN approaches ∼4×) and scripts/TPIWS.png (Figure 6: TPI= $2t$ and warp-shuffle gains on small moduli). Compare sweep shape to shipped scripts/data/1748900378.1196628.csv if not regenerating TPI plots from the new sweep file.

(E6): Section 6.1: modular multiplication latency / speedup [10 human-minutes + ∼1 compute-minute (quick subset) or ∼25 compute-minutes (full 64–4096-bit sweep), + ∼10 compute-minutes for the optional arkworks baseline]

How to: Reproduce the Section 6.1 figure comparing VROOM to BN_mod_mul_montgomery, FLINT, and arkworks.

Preparation: Build repo A (Set-up, repo A). The arkworks baseline additionally needs Rust/cargo, which ./install_deps.sh installs under $HOME via rustup; ./reproduce.sh puts it on PATH automatically, so no manual setup is required.

Execution: From the vroomssl root (all pinned to core 0):

```
./reproduce.sh modmul          # full sweep
# MODMUL_QUICK=1 ./reproduce.sh modmul
#   restricts to the 5 table sizes
./reproduce.sh arkworks        # arkworks
./reproduce.sh openssl modmul # OpenSSL
```

Results: BM_RNSRingMultiplication is VROOM; BM_BSSLModMulMontgomery is the scalar baseline; FLINT is BM_FLINTMPN<limbs> (≤1024 bits) plus BM_FLINTFMPZRing (larger); arkworks prints mul <bits> Criterion lines. For moduli ≥1500 bits, VROOM is ≈4× faster than the scalar baseline and FLINT and ≈8× faster than arkworks. (Raw bssl_bench filters are in repo A's README.md.) The OpenSSL line comes from the separate forked-

OpenSSL harness: `BM_OSSLModMulMontgomery` (the scalar `BN_mod_mul_montgomery`, essentially identical to BoringSSL) and the AVX-512 IFMA AMM points `BM_AMM52x{20,30,40}_x1_ifma256` at 1024/1536/2048 bits, over which VROOM is ≈1.7× faster for ≥1500-bit moduli.

(E7): Section 6.2: modular exponentiation and RSA-CRT (Tables 7, 9) [10 human-minutes + ∼10 compute-minutes]
How to: Reproduce the generic public-/secret-exponent exponentiation table (Table 7) and the RSA-CRT exponentiation table (Table 9).
Preparation: Build repo A (Set-up, repo A).
Execution: From the `vroomssl` root:

```
# Table 7
./reproduce.sh modexp          # VROOM
./reproduce.sh openssl modexp  # OpenSSL
# Table 9
./reproduce.sh rsacrt          # VROOM
./reproduce.sh openssl rsacrt  # OpenSSL
```

Results: Table 7 (microseconds, moduli 1024–4096): *public* $e = 65537$ rows `BM_RNSModExpRSAE` (VROOM), `BM_BSSLModExpRSAE` (BoringSSL), `BM_FLINTExponentiationRSAE` (FLINT); *secret*-exponent rows `BM_RNSExponentiation<bits,window>` (VROOM; the paper reports the fastest window per size), `BM_BSSLModExpConsttime`, and `BM_FLINTExponentiation` (FLINT, not constant-time). Table 9 (RSA-CRT, microseconds): `BM_SpeedRNSRSAModExpCRT` (VROOM) vs. `BM_SpeedRSAModExpCRT` (BoringSSL). VROOM is faster than every in-binary baseline. The OpenSSL columns are produced by the separate forked-OpenSSL harness: `BM_OSSLModExpRSAE` (public *e*), `BM_OSSLModExpConsttime` / `BM_OSSLModExpConsttime_x2` (secret *e*; the paper reports whichever is faster), and `BM_OSSLRSAModExpCRT` (RSA-CRT).

(E8): Section 6.3: RSA signing and verification (Table 8) [5 human-minutes + ∼1 compute-minute]
How to: Reproduce the RSA sign/verify latency table.
Preparation: Build repo A (Set-up, repo A).
Execution: From the `vroomssl` root:

```
./reproduce.sh rsa         # VROOM + BoringSSL
./reproduce.sh openssl rsa # OpenSSL column
```

Results: Microseconds for RSA-2048/3072/4096. VROOM rows `BM_SpeedRNSRSASign` / `BM_SpeedRNSRSAVerify`; BoringSSL rows `BM_SpeedRSASign` / `BM_SpeedRSAVerify`. VROOM signing is ≈2.3–3.2× and verification ≈3.3–3.6× faster than BoringSSL. The OpenSSL column of Table 8 is an external baseline produced by a separate forked-OpenSSL harness (`baselines/openssl`, fetched on demand as a git submodule the first time it is requested); `./reproduce.sh openssl rsa` builds it and prints the `BM_RSASign_RSA_sign` / `BM_RSAVerify_RSA_verify` rows, measuring the same `RSA_sign` / `RSA_verify` functions as BoringSSL (VROOM: RSA-4096 verify 4.0×, sign 1.3× over OpenSSL).

A.5 Notes on Reusability

The following apply to repo B.

- **Minimal API tour:** `make -C examples` and `examples/README.md` (five programs covering parameter setup, singular modmul, sum/difference/product of products).
- **Integer fallback:** `Makefile.fallback` / `FALLBACK=1` for machines without IFMA—correctness only.
- **Parameter regeneration:** `scripts/rns_secp256k1.py` and `scripts/gen_qr.py` re-emit `.hpp` parameter files; rebuild and re-run `smoke_test.sh` afterward.
- **Module map:** `src/README.md`, `cpu/`, `gpu/README.md`.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Bridging Usability and Performance: A Tensor Compiler for Autovectorizing Homomorphic Encryption

Edward Chen
Carnegie Mellon University
ejchen@cmu.edu

Fraser Brown
Carnegie Mellon University
fraserb@cmu.edu

Wenting Zheng
Carnegie Mellon University
wenting@cmu.edu

A Artifact Appendix

A.1 Abstract

This artifact contains Rotom, a compilation framework that autovectorizes tensor programs into optimized homomorphic encryption (HE) programs. The artifact includes the complete Rotom compiler implementation (approximately 12,000 lines of Python), a comprehensive suite of tensor workloads from HE compiler literature and state-of-the-art secure inference protocols, automated scripts to reproduce experiments from the paper, and documentation with setup instructions, usage examples, and guides for extending Rotom. The artifact supports the paper's main claims by enabling reviewers to compile tensor programs and verify Rotom's compilation times (Figure 6), execute compiled HE programs and measure runtime performance (Figure 7), reproduce microbenchmark results demonstrating ApplyRoll optimizations (Tables 1-3), and verify Rotom's performance against baseline compilers (Fhelipe [2] and Viaduct-HE [3]). All experiments target the CKKS implementation in OpenFHE [1] and can be reproduced on a standard Linux machine with sufficient CPU cores and memory.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical concerns associated with evaluating this artifact. All experiments run in user space within a Python virtual environment. The artifact performs purely computational tasks (tensor compilation and homomorphic encryption operations) on synthetic benchmark data. No real user data, personally identifiable information, or sensitive information is processed. The OpenFHE library used for executing compiled programs is a well-established, open-source homomorphic encryption library with no known security vulnerabilities that would affect artifact evaluation.

A.2.2 How to access

The artifact is publicly available through three channels. For reproducibility and artifact evaluation, we provide a permanent archived version on Zenodo at `https://zenodo.org/records/17957733`, which corresponds to the version of Rotom evaluated for this paper. For convenience, we provide a pre-built Docker image at `https://ghcr.io/edwjchen/ae-rotom:latest`, which includes all dependencies and baseline compilers pre-installed. For the latest updates, bug fixes, and ongoing development, the Rotom repository is available on GitHub at `https://github.com/cmu-cryptosystems/Rotom`.

A.2.3 Hardware dependencies

Rotom does not require any specialized hardware and can run on standard commodity CPUs. However, for optimal performance and to reproduce the paper's results, we recommend a multi-core processor with at least 16 cores (64 cores recommended for full benchmarks), 256 GB RAM required for larger benchmarks, and at least 10 GB of free disk space. The evaluation in the paper was conducted on a 64-core Intel Xeon CPU with 256 GB of RAM.

A.2.4 Software dependencies

Rotom was tested on Ubuntu 24.04 LTS. The artifact should work on other modern Linux distributions with minimal modifications. The core software dependencies are Python 3.12, NumPy for numerical operations, and OpenFHE for executing compiled homomorphic encryption programs. All Python dependencies, including specific versions of NumPy and other required packages, are specified in the `requirements.txt` file included in the artifact and can be installed automatically using `pip`. OpenFHE is an open-source fully homomorphic encryption library available at `https://github.com/openfheorg/openfhe-development`, and installation instructions are provided in the artifact documentation.

Docker (recommended): To simplify dependency installation and baseline setup, we provide a pre-built Docker image.

Pull and run the image with:

```
docker pull ghcr.io/edwjchen/ae-rotom:latest
docker run -it ae-rotom bash
```

The working directory inside the container is `/app/Rotom`.

A.2.5 Benchmarks

The benchmark suite consists of six application-level tensor workloads and three microbenchmarks, all of which are included directly in the artifact. All benchmarks use synthetic data.

Application-level benchmarks: (1) MatMul performs ciphertext-plaintext matrix multiplication, (2) Double-MatMul performs two consecutive ciphertext-ciphertext matrix multiplications, (3) TTM computes the third-order tensor matrix product, (4) Convolution performs 2D convolution over multi-channel images, (5) LogReg-MatVecMul performs two consecutive matrix-vector multiplications for logistic regression, and (6) BERT Attention implements the attention layer from the BERT-base transformer model.

Microbenchmarks: The artifact includes three targeted microbenchmarks that evaluate the performance of ApplyRoll and its optimized variants (ApplyRotRoll, ApplyBSGSRoll, ApplyBSGSMatMul) compared to ApplyPermute, as well as the effectiveness of Strassen's algorithm.

Baseline compiler benchmarks: To compare Rotom's compile time against Fhelipe and Viaduct-HE, the artifact requires both Fhelipe and Viaduct-HE. These repositories can be found through https://github.com/fhelipe-compiler/fhelipe and https://github.com/rolph-recto/viaduct-he/.

The artifact provides another directory containing Rotom's benchmarks that can eb copied into each of these systems. This directory contains instruction traces for various benchmark operations from both baseline systems. Some traces have been hand-modified to ensure compatibility with Rotom's wrapper interface and to enable execution on Rotom's OpenFHE backend. Installation instructions for these benchmarks are provided in the directory. An online version of these benchmarks can also be found here: https://github.com/edwjchen/rotom_benchmarks.

A.3 Set-up

Option A: Docker (recommended). Pull and launch the prebuilt image:

```
docker pull ghcr.io/edwjchen/ae-rotom:latest
docker run -it ae-rotom bash
```

Inside the container, run the provided setup script to clone and copy benchmarks and scripts into the correct locations:

```
/app/setup.sh
```

The working directory is `/app/Rotom`. Proceed directly to the Basic Test below.

Option B: Manual installation. Clone the Rotom repository, create a Python virtual environment, and install dependencies using `pip install -r requirements.txt`.

A.3.1 Basic Test

Run `pytest` in the Rotom directory to execute all unit tests. Run `python main.py` to compile and execute a basic matrix multiplication benchmark.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Rotom achieves fast compilation times, compiling all tensor workloads in under 5 minutes (most in seconds). This is proven by experiment (E1) described in Section 8.2.1, whose results are reported in Figure 6.

(C2): Rotom achieves high search quality, matching or outperforming state-of-the-art HE compilers (Fhelipe and Viaduct-HE) by up to 80× in execution time. This is proven by experiment (E2) described in Section 8.2.2, whose results are reported in Figure 7.

(C3): Rotom's ApplyRoll operator provides significant performance improvements over ApplyPermute for layout conversions, achieving 3-8× speedup in common scenarios. This is proven by experiments (E3) described in Section 8.1, whose results are reported in Tables 1-3.

A.4.2 Experiments

(E1): *[Compilation Time] [30 human-minutes + 48 compute-hours]: Measure Rotom's compilation time for all benchmarks and compare against Fhelipe and Viaduct-HE to validate claim (C1).*

Preparation: If using Docker, run `/app/setup.sh` inside the container; this clones `rotom_benchmarks`, Fhelipe, and Viaduct-HE and copies all benchmarks and scripts into the correct locations automatically. If building manually, ensure all dependencies are installed for Rotom by following its README, then build Fhelipe (https://github.com/fhelipe-compiler/fhelipe) and Viaduct-HE (https://github.com/rolph-recto/viaduct-he/) following their respective instructions. Copy `rotom_benchmarks/fhelipe_apps` into `fhelipe/frontend/fheapps/rotom` and `rotom_benchmarks/viaduct_apps` into `viaduct-he/benchmarks`.

Execution: Run `python run_all_tests.py` to compile (and execute) all benchmarks with Rotom, Fhelipe, and Viaduct-HE at both $n = 8K$ and $n = 32K$. The script automatically records compilation times of Rotom and

saves results to `Rotom/logs`. To compare against Fhelipe and Viaduct-he, we provide compilation scripts within `rotom_benchmarks/scripts` and they are intended to be run within the top-level directory of each system. Note that each of the bert-attention benchmarks takes around 30 minutes to an hour to run.

Results: Use `python parse_logs.py` to parse all of Rotom's logs and generate a plot of Rotom's compile time.

(E2): *[Runtime Performance] [10 human-minutes + 48 compute-hours]: Measure execution time of compiled HE programs from Rotom, Fhelipe, and Viaduct-HE to validate claim (C2).*

Preparation: If using Docker, run `/app/setup.sh` inside the container; as stated in (E1). If building manually, first ensure that all instruction traces are within `Rotom/benchmarks`. These traces are found in https://github.com/edwjchen/rotom_benchmarks. Both `fhelipe_benchmarks` and `viaduct_benchmarks` need to be copied to `Rotom/benchmarks`. Additionally, ensure all Python dependencies are installed. Please run `pytest` to execute all unit tests for Rotom.

Execution: Run `python run_all_tests.py` to execute all compiled programs on the OpenFHE backend. The script runs each benchmark 5 times and records average execution time, communication costs (LAN/WAN), and operation counts. Results are saved to `Rotom/logs`. Note that each of the bert-attention benchmarks takes around 30 minutes to an hour to run.

Results: After `python run_all_tests.py` has finished, use `python parse_logs.py` to parse all of Rotom's logs. This script should generate both compile and execution times for Rotom (and other systems) as a plot in `Rotom/plots`.

(E3): *[Microbenchmarks] [5 human-minutes + 10 compute-minutes]: Evaluate ApplyRoll optimizations and compare against ApplyPermute to validate claim (C3).*

Preparation: No additional preparation required beyond basic installation.

Execution: Run `./scripts/rotom/benchmark_roll_conversion.sh`, `./scripts/rotom/benchmark_slot_roll_conversion.sh` and `./scripts/rotom/benchmark_strassens.sh` to execute three microbenchmarks: (1) swapping two dimensions, (2) swapping two slot dimensions with BSGS, and (3) Strassen's algorithm with different conversions. Results are saved to `Rotom/logs/*`, where * is the name of the script (without `benchmark_`).

Results: Compare operation counts (Add, Mul, Rot) and execution times in the respective logs.

Note: Compute time estimates assume a 64-core machine with 256 GB RAM. Machines with fewer resources may require proportionally longer execution times, particularly for larger benchmarks.

A.5 Notes on Reusability

Rotom is designed to be extensible and reusable for a variety of homomorphic encryption research and applications beyond the experiments in this paper.

Writing custom tensor programs: Users can write new tensor programs using Rotom's DSL, which closely resembles PyTorch. The `frontends/tensor.py` module provides common operators (MatMul, Conv2D, Add, etc.) that can be composed to build custom workloads. Examples of tensor programs are provided in `benchmarks/` and can serve as templates for new applications.

Integrating with other HE backends: While Rotom currently targets OpenFHE, it can be extended to support other HE libraries by implementing a new backend in `backends/`. The toy backend (`backends/toy.py`) serves as a plaintext example for correctness testing and can used as a reference for other frameworks.

Visualizing layouts: Rotom includes a layout visualization tool (`layout_visualizer.py`) that helps understand how tensors are packed into ciphertexts, useful for debugging and educational purposes.

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/usenixsec2026/.

References

[1] Ahmad Al Badawi, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, et al. Openfhe: Open-source fully homomorphic encryption library. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography*, pages 53–63, 2022.

[2] Aleksandar Krastev, Nikola Samardzic, Simon Langowski, Srinivas Devadas, and Daniel Sanchez. A tensor compiler with automatic data packing for simple and efficient fully homomorphic encryption. *Proceedings of the ACM on Programming Languages*, 8(PLDI):126–150, 2024.

[3] Rolph Recto and Andrew C Myers. A compiler from array programs to vectorized homomorphic encryption. *arXiv preprint arXiv:2311.06142*, 2023.



USENIX Security '26 Artifact Appendix: <Fend for Yourself! Backdoor Purification in Federated Graph Learning with an Evolving Knowledge Anchor>

Chengcheng Zhu
Nanjing University

Yunlong Mao *
Nanjing University

Jiale Zhang
Yangzhou University

Bosen Rao
Yangzhou University

Sheng Zhong
Nanjing University

A Artifact Appendix

A.1 Abstract

This appendix describes the software artifact that implements the algorithms and evaluations presented in the paper "Fend for Yourself! Backdoor Purification in Federated Graph Learning with an Evolving Knowledge Anchor". Specifically, the artifact package includes the complete source code of GBHINDER, along with the baseline methods used for comparison in the experiments, including multiple backdoor attack strategies and defense mechanisms in Federated Graph Learning (FedGL). The implementation supports a variety of benchmark graph datasets and several graph neural network models. Additionally, the artifact provides shell scripts to facilitate the reproduction of experiments and a `README.md` file offering detailed instructions for setup and usage.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The primary focus of our research is purely defensive, designed to safeguard FedGL models against malicious backdoor attacks. Although this artifact includes implementations of attack methodologies, they are provided solely for the purpose of evaluating defense mechanisms and fostering further research into robust federated learning. Consequently, the ethical implications of this work are positive, as it aims to secure technologies frequently deployed in safety-critical domains such as finance and healthcare. Furthermore, we exclusively utilized public, open-source benchmark datasets, ensuring that no private or personally identifiable information was handled or compromised during our research.

*Corresponding author, email: maoyl@nju.edu.cn.

A.2.2 How to access

The source code and datasets are available on GitHub (https://github.com/CBond00/GraphFL-defense) and Zenodo (https://doi.org/10.5281/zenodo.17898577).

A.2.3 Hardware dependencies

The artifact requires the following operating system and hardware resources:

- **CPU:** AMD Ryzen 9 9950X.
- **RAM:** 32 GB of system memory.
- **GPU (optional):** The experiments were conducted on an NVIDIA GeForce RTX 5080 GPU. Other comparable or higher-end consumer-grade GPUs are also acceptable.
- **Storage:** Sufficient disk space is required to download benchmark datasets and to store trained models, intermediate checkpoints, and experiment logs.

The specifications listed above correspond to the hardware configuration used in our experiments. Similar or more powerful hardware configurations are also suitable for running the artifact.

A.2.4 Software dependencies

The artifact requires the following operating system and essential software packages:

- **OS:** Windows or Linux.
- **Python Version:** Python 3.9.
- **Packages:** PyTorch, Pytorch_geometric, Pytorch_sparse, NetworkX, NumPy, Matplotlib, Scikit-learn, Hdbscan, and PyYAML.

Please refer to the instructions provided in the `README.md` file in our repository for detailed environment setup and dependency installation guidelines.

A.2.5 Benchmarks

We mainly conduct experiments on 4 public real-world datasets: NCI1, PROTEINS, DD, and AIDS. Other datasets, such as MUTAG and COLLAB, are also supported. These datasets are automatically downloaded during the first run of the script.

A.3 Set-up

Please verify that the hardware and software dependencies are satisfied and download the source code from the repository. The environment required for running this artifact is specified in the `requirements.txt` file. Please ensure that torch is the GPU version and that the version matches the geometric library. To set up the environment, open a terminal and execute the following commands:

```
conda create -n FedGLenv python=3.9
conda activate FedGLenv
conda install conda-forge::pytorch-gpu
conda install conda-forge::pytorch_geometric
conda install conda-forge::pytorch_sparse
pip install -r requirements.txt
```

A.3.1 Basic Test

The test runs a short federated learning session with a backdoor attack enabled over a limited number of rounds.

```
python main.py --dataset AIDS --model GIN --
    ↪ start_round 0 --end_round 600 --
    ↪ no_of_total_participants 10 --
    ↪ no_of_participants_per_round 5 --
    ↪ resumed_model False --attack_method
    ↪ motif --no_of_adversaries 2 --
    ↪ poisoned_start_round 500 --
    ↪ poisoned_end_round 600 --injection_ratio
    ↪  0.5 --trigger_ratio 0.2 --trigger_num 3
    ↪  --defense_method nodefense --benign_lr
    ↪ 0.01 --poisoned_lr 0.01
```

You can also run an equivalent instance directly using the following command.

```
python test_quick.py
```

If the artifact is set up correctly, the following behavior should be observed: The specified dataset is automatically downloaded to the `./data` directory if it does not already exist. A new directory `saved_models/{timestamp}/` is created, containing `log.txt` and `params.yaml`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Attackers need only participate for a limited number of rounds (e.g., 10 rounds) to implant a backdoor; significantly, this backdoor effect remains persistent even after the adversaries exit the aggregation process, and the FL system continues with only benign participants. This is substantiated by experiment (E1), as detailed in Section 4 of the paper, with results presented in Figure 1.

(C2): GBHINDER is able to effectively defend against a wide range of backdoor attack methods across multiple benchmark datasets, significantly outperforming existing baseline defense approaches. This claim is supported by the experiment (E2), with the corresponding results reported in Table 1.

(C3): GBHINDER remains effective under different adversarial constraints, including varying poisoning ratios, proportions of malicious participants, and heterogeneous data distributions. This is validated by experiment (E3), with results presented in Figures 3 and 4.

A.4.2 Experiments

(E1): *[Backdoor attack effectiveness] [10 human-minutes + 1 compute-hour + 1GB disk]:* This experiment validates the effectiveness of various backdoor attacks in the absence of defense mechanisms. Specifically, it demonstrates that the backdoor effect remains persistent within the federated learning process even after the active attack phase has ceased.
How to:
Preparation: Ensure that the Conda environment is created and activated. Upon the first execution of the code, the datasets corresponding to the specified command will be automatically downloaded to the `./data` directory.
Execution: To accelerate the experimental process, first train a clean pre-trained global model. This model can be loaded directly in subsequent experiments to skip the initial training phase.

```
python main.py --dataset AIDS --model GIN --
    ↪ start_round 0 --end_round 500 --
    ↪ no_of_total_participants 10 --
    ↪ no_of_participants_per_round 5  --
    ↪ defense_method nodefense --
    ↪ resumed_model False --benign_lr 0.01
    ↪  --GPU_id 0
```

After obtaining the clean model, you can set `-resumed_model` to `True` (or ignore the pre-trained model to start from scratch). Configure the attack strategy (e.g., `motif`), the number of adversaries (e.g., `2`), and other attack parameters. Set the attack window to rounds 500–600, and extend the total training duration to 1000 rounds to observe the persistence of the backdoor.
Results: The datasets are saved in `./data`. After experiments complete, results will be saved in the `saved_models/timestamp/` directory: `global.csv`, `log.txt`, `params.yaml`, and `saved_model_*.pt.tar`.

(E2): *[Defense effectiveness] [10 human-minutes + 5 compute-hour + 5GB disk]:* This experiment evaluates

the performance of varying defense mechanisms against different attack strategies across multiple datasets. It compares GBHINDER against the baseline defense methods presented in Table 1 of the paper.

How to:

Preparation: Ensure that the Conda environment is created and activated. If a clean pre-trained global model is required, please follow (E1) to obtain the checkpoint and reuse it for the following evaluations.

Execution: To run the complete comparative evaluation automatically, execute the provided shell script:

```
python test_full_evaluation.py
```

Alternatively, to evaluate arbitrary combinations of attack and defense strategies, you can execute the main script manually. Please refer to the `README.md` for detailed configuration options.

```
python main.py --dataset AIDS --model GIN --
    ↪ start_round 0 --end_round 1000 --
    ↪ no_of_total_participants 10 --
    ↪ no_of_participants_per_round 5  --
    ↪ attack_method [optional] --
    ↪ no_of_adversaries 2 --
    ↪ poisoned_start_round 500 --
    ↪ poisoned_end_round 600 --
    ↪ injection_ratio 0.5 --trigger_ratio
    ↪ 0.2 --trigger_num 3 --defense_method
    ↪  [optional] --GPU_id 0
```

The optional fields can be replaced with the desired backdoor attack method and defense method supported by the artifact.

Results: The output format and directory structure are consistent with (E1).

(E3): *[Defense robustness] [10 human-minutes + 5 compute-hour + 5GB disk]:* This experiment assesses the robustness of the defense mechanisms under varying adversarial settings. Specifically, it evaluates performance across a range of poisoning ratios $(0.1, 0.3, 0.5, 0.7, 0.9)$ and varying proportions of adversaries $(10\%, 20\%, 30\%, 40\%, 50\%)$.

How to:

Preparation: Ensure that the Conda environment is created and activated.

Execution: To run the full robustness benchmarks, execute the following scripts:

```
python test_injection_ratio.py
python test_client_num.py
```

Alternatively, to manually test specific poisoning ratios or adversary counts, refer to the execution configuration in (E2) and modify the `-injection_ratio` and `-no_of_adversaries` parameters accordingly.

Results: The output format and directory structure are consistent with (E1).

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Breaking the Boundaries: Analyzing QUIC Frame-Packet Interactions With QUIC-Attacker

Nurullah Erinola, Marcel Maehren, Marcus Brinkmann, and Jörg Schwenk

Ruhr University Bochum

A Artifact Appendix

A.1 Abstract

In our paper, we developed probes to explore how different QUIC server implementations handle the coalescence and fragmentation of payloads, covering both valid and invalid combinations of datagrams, packets, and frames. Already at this basic level, we observe significant differences between implementations, some of which pointing towards exploitable vulnerabilities. Previous QUIC research tools were not designed to implement such probes. To address this limitation, we presented *QUIC-Attacker*, a testing framework that provides maximum freedom on the sending side of QUIC.

This artifact appendix aims to show the functionality of *QUIC-Attacker*, which we developed to conduct our study. The experiments involve the reproduction of our most important findings, including the eight DoS vulnerabilities caused by unhandled exceptions and the exploitable injection vulnerabilities in Kwik and XQUIC.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We are not aware of any security, privacy, or ethical concerns arising from running the experiments described below.

A.2.2 How to access

We provide our artifacts via Zenodo.[1]

A.2.3 Hardware dependencies

At least 60 GB of free disk space is required to build Docker images.

A.2.4 Software dependencies

The experiments require the ability to run bash scripts and basic Linux CLI tools (curl, unzip, and grep). In addition, Python 3 and Docker must be installed as part of the setup. We tested this artifact on Ubuntu 24.04 LTS Desktop.

[1] https://doi.org/10.5281/zenodo.21451785

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Download and unzip the `artifact_experiments.zip` achieve from the Zenodo repository:

https://doi.org/10.5281/zenodo.21451785

Then, run the `setup.sh` script from the root directory of the extracted artifact experiments directory:

```
./setup.sh all
```

The script performs the following steps:

1. Downloads all remaining archives from Zenodo.
2. Unzips the downloaded archives.
3. Builds QUIC-Attacker and the test suite.
4. Builds the Docker images required for the experiments.

A.3.2 Basic Test

Testing the QUIC-Attacker. To verify that QUIC-Attacker can be executed correctly, run the following command:

```
./setup.sh test-attacker
```

If everything works correctly, this should print the parameter list of QUIC-Attacker with usage instructions.

Testing the Test Suite. To verify that the test suite can be executed correctly, run the following command:

```
./setup.sh test-suite
```

If everything works correctly, this should print the parameter list of the test suite with usage instructions.

A.4 Evaluation workflow

A.4.1 Major Claims

We claim to provide QUIC-Attacker, which can be used to evaluate various QUIC implementations, especially with respect to coalescing and fragmentation. In addition, we claim to provide the QUIC-Docker-Library, which can be used to easily start different QUIC implementations with Docker. In the following, we exemplary describe our claims for our most important findings: eight DoS vulnerabilities caused by unhandled exceptions and two exploitable injection vulnerabilities.

(C1): We claim to have found eight crashes in three QUIC server implementations (LSQUIC, Neqo, and quiche^C). These crashes are described in Sections 7.2.3 and 7.3 of our main paper.

(C2): We claim to have found exploitable injection vulnerabilities in two QUIC server implementations (XQUIC and Kwik). These vulnerabilities are described in Sections 7.2.1 and 7.2.2 of our main paper.

(C3): We claim to provide a proof of concept exploit for the injection vulnerability in XQUIC (cf. Section 7.2.1 and Figure 4 of our main paper).

A.4.2 Experiments

(E1): *Executing the DoS analysis [10 human-minutes]: this experiment executes the required QUIC handshakes with QUIC-Attacker against three QUIC implementations to reproduce the presented crashes.*

Preparation: Run the `E1.sh` script from the root directory of the extracted artifact experiments directory:

```
./E1.sh start
```

The script starts the three QUIC servers.

Execution: Run the `E1.sh` script again to open the logs of the QUIC servers:

```
./E1.sh logs
```

Note that the logs are intentionally reduced to make the crashes easier to detect. In a second terminal, use the `E1.sh` script again to execute the required QUIC handshakes. Each of the following commands executes one QUIC handshake and must be run sequentially. The table below also maps each command to the tested implementation and the corresponding test identifier used in our main paper:

Command	Test	Implementation
`./E1.sh exp-1.1`	F7	Neqo
`./E1.sh exp-1.2`	S1	Neqo
`./E1.sh exp-1.3`	S2	Neqo
`./E1.sh exp-1.4`	S3	Neqo
`./E1.sh exp-1.5`	S1	quiche^C
`./E1.sh exp-1.6`	S2	quiche^C
`./E1.sh exp-1.7`	S3	quiche^C
`./E1.sh exp-1.8`	S3	LSQUIC

Results: For each execution, inspect the logs. Each executed QUIC handshake crashes the tested server and triggers a restart. The first four executions trigger crashes of Neqo, the next three executions trigger crashes of quiche^C, and the final execution triggers a restart of LSQUIC. Since the experiment consists of eight handshakes, you should observe eight restarts in total.

(E2): *Executing the injection vulnerability analysis [10 human-minutes]: this experiment executes a minimum set of required QUIC handshakes with QUIC-Attacker against two QUIC implementations to demonstrate the missing frame-packet validation.*

Preparation: Run the `E2.sh` script from the root directory of the extracted artifact experiments directory:

```
./E2.sh start
```

The script starts the two QUIC servers.

Execution: Use the `E2.sh` script again to execute the minimum set of required QUIC handshakes. Each command evaluates one QUIC implementation and must be run sequentially. The table below also maps each command to the tested implementation and the corresponding test identifier used in the main paper:

Command	Test	Implementation
`./E2.sh exp-2.1`	F1 & F4	Kwik
`./E2.sh exp-2.2`	F1 & F4	XQUIC

Results: Each execution performs four QUIC handshakes with the tested server.[2]

For Kwik, the script should produce the following:

```
1 FrameTests - INITIAL_PACKET(MAX_STREAM_DATA_FRAME): CC
2 FrameTests - HANDSHAKE_PACKET(MAX_STREAM_DATA_FRAME): CC +
      ↪ TLS FIN
3 FrameTests - INITIAL_PACKET(STREAM_FRAME): DONE + TLS FIN
4 FrameTests - HANDSHAKE_PACKET(STREAM_FRAME): DONE + TLS FIN
```

Each line summarizes the server's response to one tested frame-packet combination. All four combinations are forbidden, and a QUIC implementation must treat them as protocol violations by terminating the connection. However, Kwik terminates the connection with a `CC (CONNECTION_CLOSE)` frame only for the first two combinations. For the other two combinations, the handshake completes without such a frame. This behavior indicates missing frame-packet validation.

For XQUIC, the script should produce the following:

```
1 FrameTests - INITIAL_PACKET(MAX_STREAM_DATA_FRAME): DONE +
      ↪ TLS FIN
2 FrameTests - HANDSHAKE_PACKET(MAX_STREAM_DATA_FRAME): DONE +
      ↪ TLS FIN
3 FrameTests - INITIAL_PACKET(STREAM_FRAME): DONE + TLS FIN
4 FrameTests - HANDSHAKE_PACKET(STREAM_FRAME): DONE + TLS FIN
```

[2] A complete analysis requires testing all frame-packet combinations described in the main paper. For the artifact evaluation, we reduced the analysis to four combinations because these are sufficient to observe a deviation from the specification.

In all four combinations, XQUIC neither sends a `CC (CONNECTION_CLOSE)` frame nor otherwise terminates the connection. This behavior also indicates missing frame-packet validation.

(E3): *Executing the proof of concept exploit [5 human-minutes]: this experiment executes the presented confusion attack against the XQUIC implementation.*

Preparation: The Chrome container in this experiment needs access to the host's X server to display its graphical interface. For this purpose, grant local Docker containers access to the X server by running:

```
xhost +local:docker
```

Execution: Run the `E3.sh` script from the root directory of the extracted artifact experiments directory:

```
./E3.sh start
```

The script start the XQUIC server, the Chrome browser, and the attacker script. Note, that Chrome opens an initial welcome window, which can be skipped. It then automatically issues a `GET` request to `/cat` on the server.

Results: Due to the injected frame by the attacker, the response delivered to the browser corresponds to `/dog` instead. This is visible in the browser, which displays the content `DOG DOG DOG...` instead of the expected `CAT CAT CAT...`.

A.5 Notes on Reusability

We contribute QUIC-Attacker as part of the TLS-Attacker suite (https://github.com/tls-attacker/TLS-Attacker). Our extensions have been merged and released as part of TLS-Attacker v7.8.0 (https://github.com/tls-attacker/TLS-Attacker/releases/tag/v7.8.0). We also provide the QUIC-Docker-Library on GitHub (https://github.com/tls-attacker/QUIC-Docker-Library). Both projects will be maintained and extended in their respective repositories. For updates, please refer to the GitHub repositories.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Secure Protocol Composition under Dynamic Corruption: Scaling Up Symbolic Analysis for Real-World Security Properties

Cas Cremers
CISPA Helmholtz Center for Information Security

Erik Pallas
CISPA Helmholtz Center for Information Security

Aleksi Peltonen
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

Although automated symbolic protocol verification has proven valuable and effective, current approaches begin to reach their limits: While small protocols can be analyzed automatically, the most complex case studies often require substantial expert time and resources. There have been many attempts to solve this problem by compositional verification, but they rely on unrealistic protocol assumptions and do not support real-world security properties like Forward Secrecy.

In this work, we enable compositional symbolic analysis for real-world security protocols with respect to modern security properties. We develop a composition result in the Applied π-Calculus that holds even in the presence of attackers capable of dynamic corruption if the protocols satisfy a disjointness requirement. We demonstrate the applicability and effectiveness of our result on the composition of a data exchange protocol with a Diffie-Hellman key exchange and a compositional analysis of Forward Secrecy in TLS 1.3 ECH.

This artifact contains the ProVerif models of both our case studies and the Full Version of our paper, including all proofs which did not fit into the space constraints of the conference version.

A.2 Description & Requirements

The artifact consists of a set of ProVerif models and the Full Version of our paper containing proofs and additional theory. Since the latter is a simple, self-contained PDF, the remainder of this Artifact Appendix refers to the ProVerif models unless specifically stated otherwise.

We provide models for two case studies. The first consists of an authenticated Diffie-Hellman key exchange and a simple data transfer protocol to demonstrate at a small scale how a compositional analysis can be set up in ProVerif. The second performs both a compositional and a monolithic analysis of Forward Secrecy in TLS 1.3 ECH while benchmarking and comparing the resource consumption of both types of analysis. It builds on prior work by Bhargavan, Cheval and Wood (cf. Karthikeyan Bhargavan, Vincent Cheval, and Christopher Wood. 2022. A Symbolic Analysis of Privacy for TLS 1.3 with Encrypted Client Hello. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM, New York, NY, USA, 365–379. doi:10.1145/3548606.3559360)

A.2.1 Security, privacy, and ethical concerns

There are no security or privacy risks associated with running the ProVerif models of our case studies. We also see no related ethical concerns; for a detailed analysis, see the Ethical Considerations appendix of our main paper.

Note, however, that a full benchmark of the TLS 1.3 ECH case study is very memory-intensive and can take up to several days. We therefore recommend to not run the benchmark on hardware used for day-to-day business but use a virtual machine, a remote server or similar instead.

A.2.2 How to access

The artifact is hosted in the Figshare repository identified by DOI 10.6084/m9.figshare.29958290. It contains two files:

- `secure_composition_under_dynamic_corruption_case_studies.pdf`: The Full Version of our paper, containing all proofs and theory omitted from the conference version due to space constraints.
- `secure_composition_under_dynamic_corruption_case_studies.zip`: An archive containing all models and build files needed for our case studies.

To recreate our experiments, download and unpack the `.zip` archive. All file paths in the following are relative to the root of this unpacked archive.

A.2.3 Hardware dependencies

There are no special hardware requirements to run the ProVerif models in this artifact. ProVerif itself requires

OCaml to be installed, which does not have official hardware requirements. However, we recommend to provide at least 200 GB of memory since benchmarking the TLS 1.3 ECH case study is memory-intensive. Alternatively, if omitting the scenarios that will likely time out after 48 hours, 70 GB of memory will suffice. For reference, we performed the benchmarks on a server featuring an Intel® Xeon® CPU E5-4650L with 2.60 GHz and 1 TB of RAM.

A.2.4 Software dependencies

The artifact requires the `make` build management tool to build the models for the TLS 1.3 ECH case study, ProVerif v2.05 or later to analyze the models, and OCaml and/or `opam` to install ProVerif. If you use the provided Dockerfile to run ProVerif, you also need a compatible container engine like Docker or Podman. At the time of this writing, ProVerif v2.05 is the most recent version; we also recommend using v2.05 after possible future releases to maintain consistency of the results.

A.2.5 Benchmarks

All models and benchmarking scripts are provided with the artifact.

A.3 Set-up

The following has been tested on Ubuntu 24.04. We assume that both `make` and Docker are already installed. For a detailed explanation of the structure of the case study files, see `\tls_ech\README.md`.

A.3.1 Installation

All dependencies can be either installed directly on a machine, or you can use the provided Dockerfile to run the case studies in a container.

ProVerif can be either installed from source or via the OCaml Package Manager `opam`; we recommend the latter. If not already present, install `opam` using the package manager of your choice (e.g., `apt install opam`) or follow the official installation instructions. Then, run `opam depext conf-graphviz` to install graphviz, `opam depext proverif` to install the remaining dependencies of ProVerif, and finally `opam install proverif.2.05` to install ProVerif v2.05.

If you prefer to use a container, navigate to the root directory of the unpacked artifact; there, you find a Dockerfile and an archive `proverif2.05.tar.gz` which contains the ProVerif source code. From there, run `docker build -t composition:latest -f ./Dockerfile .` to build the Docker image, and afterwards `docker run -it composition:latest` to run the resulting container. If you use a container engine other than Docker, use the corresponding `build` and `run` commands instead. Note that the Dockerfile contains instructions for installing `vim` and `emacs` for convenient inspection of text files within the container.

A.3.2 Basic Test

To check whether ProVerif is properly installed, run `proverif -help`. The resulting output should start with `Proverif 2.05. Cryptographic protocol verifier, by Bruno Blanchet, Vincent Cheval, and Marc Sylvestre` followed by explanations for several input parameters. For testing the TLS 1.3 ECH benchmark navigate to `\tls_ech` and run `bash run_bench one pfs 2 2 2 2` to perform a monolithic analysis of the first of the 32 scenarios considered, and `bash run_bench -c one pfs 2 2 2 2` for a compositional analysis. The four digits at the end of the command indicate the scenario code and can be used to separately analyze individual scenarios (for an overview of relevant scenarios, see Appendix E in our paper). On a sufficiently powerful machine the analyses of scenario 2222 will take around 10 and 5 minutes, respectively.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): All requirements of a compositional analysis of an authenticated Diffie-Hellman key exchange and a simple data exchange protocol are met, and it yields the same results as a monolithic analysis.

(C2): Compositional analysis of Forward Secrecy of the master secret in TLS 1.3 ECH decreases resource consumption by up to two orders of magnitude in ProVerif.

A.4.2 Experiments

(E1): [5 human-minutes + 100 compute-milliseconds + 10MB disk]: Compositional and monolithic analysis of the Diffie-Hellman/Data Exchange case study; cf. Section 6.2 of the paper. Validates (C1).

How to: Perform a monolithic and a compositional analysis of the Diffie-Hellman/Data Exchange case study; Forward Secrecy of the session keys will be proven true in both analyses.

Preparation: We recommend to open two terminal windows side-by-side, although this not strictly necessary. Navigate to
`dh` in both or the single terminal.

Execution:
1. If you use two terminal windows, perform this step in the first. Run `bash run_bench` for a monolithic analysis.
2. In `dh_auth.pv`, comment out lines 121 and 133; this is done in ProVerif by surrounding the line with `(* ... *)`.

3. If you use two terminal windows, perform this step in the second. Run `bash run_bench` again for a compositional analysis.

Results: ProVerif outputs some proof information followed by the results of five so-called queries in a block titled 'Verification summary'. These correspond to the theorems we want to proof.

The first two queries are sanity checks whether the modeled protocols can actually be executed; in the monolithic analysis both should evaluate to `false`, while in the compositional analysis the first should evaluate to `false` and the second to `true` (note that for these two queries, ProVerif attempts to prove the negated statement here as indicated by the `not` directly following the `Query` keyword, meaning that `false` actually indicates a successful proof).

The third query verifies the *set-up guarantee* of the Diffie-Hellman part of the model, which needs to hold for a compositional analysis. In both analyses, this should evaluate to `true`.

The fourth and fifth queries state Forward Secrecy of the parties' sessions keys generated by the Diffie-Hellman key exchange. These, too, should evaluate to `true` in both analyses.

(E2): [30 human-minutes + 460 compute-hours + 200GB disk]: Benchmarks of a compositional and monolithic analysis of Forward Secrecy in TLS 1.3 ECH; cf. Section 6.3 of the paper. Validates (C2).

How to: Run the provided benchmarking script and compare the measured resource consumption.

Preparation: Navigate to `\tls_ech`. If you have run the script before, remove all temporary files and directories (`\tls_ech\generated_configs`, `\tls_ech\generated_libraries`, `\tls_ech\tests`, `\tls_ech\ongoing.txt`) and the files containing the results (`\tls_ech\security_properties \forward_secrecy\result[_composition].txt`); otherwise, the script will skip some or all of the scenarios and not produce new data.

Execution:
1. Run `bash run_bench -c reach pfs` for the compositional analysis. On our reference hardware, this takes roughly 53 h.
2. Run `bash run_bench reach pfs` for the monolithic analysis. On our reference hardware, this takes roughly 405 h, or about 213 h when ignoring scenarios that time out.

Note that `run_bench` evaluates all scenarios sequentially and does not do any parallelization. It is possible to evaluate scenarios in parallel by analyzing them individually as described in Appendix A.3.2; if you attempt this, make sure to provide the individual processes with sufficient resources.

Results: In both cases, the benchmarking script outputs a `.txt` file containing the resource consumption. The compositional analysis results are written to `\tls_ech\security_properties \forward_secrecy\result_composition.txt`, the results of the monolithic analysis to `result.txt` in the same directory.

The results of our own benchmarking can be found in `\tls_ech\pfs_results`.

A.5 Notes on Reusability

The composition mode we added to the TLS 1.3 ECH models can be used to analyze other security properties than Forward Secrecy. However, this requires additional theoretical work to verify that the respective properties can be defined over one of the two parts in which we split the protocol, and that the requirements of the composition theorem we describe in our paper still hold. The protocol can also be split up at another position depending on the property and whether the requirements still hold.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: <Semantics Over Syntax: Uncovering Pre-Authentication 5G Baseband Vulnerabilities>

Qiqing Huang

A Artifact Appendix

A.1 Abstract

This artifact provides an end-to-end implementation of CONSET (Constraint-Guided Semantic Testing), a framework for systematically extracting specification-level constraints from 3GPP standards and generating targeted test cases to uncover semantic inconsistencies in 5G UE implementations.

The artifact consists of three main components: **(A1)** a simulation testbed providing a pre-configured Docker environment and remote access for executing experiments; **(A2)** a constraint-driven toolchain comprising 26 executable scripts that extract semantic constraints from 3GPP specifications, normalize them into DSL rules using LLM-based reasoning, and generate syntactically valid test cases that deliberately violate these constraints; and **(A3)** a collection of test cases and proof-of-concept exploits that reproduce crashes in the OpenAirInterface (OAI) UE within the simulation environment.

The artifact includes pre-processed 3GPP specification documents and enables evaluators to: (i) re-run the complete constraint extraction pipeline from specifications to DSL rules, (ii) generate test cases based on extracted constraints, and (iii) reproduce the 29 unique crash sites reported in the paper on the OAI UE. Note that portions of the constraint extraction pipeline that rely on Mathpix and OpenAI APIs are marked as optional and can be skipped without affecting the core evaluation workflow. All proof-of-concept exploits are limited to the simulation environment; over-the-air exploits are deliberately withheld to prevent misuse.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Execution Environment: The artifact is designed to run within an isolated Docker container, which minimizes risks to the evaluator's host system. Local execution of the A1/A3 Docker-based testbed requires elevated Docker privileges (e.g., `-privileged`, `-ipc=host`, `-network=host`) as documented in the Set-up section; however, all experiments are confined to an isolated simulation environment and do not interact with real networks or devices.

API Usage (Optional): Portions of the constraint extraction pipeline optionally use the OpenAI API and Mathpix API. If evaluators choose to use their own API keys, they should be aware that: (i) specification text excerpts from 3GPP documents will be transmitted to these external services for processing, and (ii) API usage will incur costs based on the providers' pricing models. Evaluators may skip these optional steps and use pre-computed intermediate results provided in the artifact.

Exploit Code: The artifact includes proof-of-concept exploit code that triggers crashes in the OpenAirInterface UE. This code is strictly limited to the simulation environment and cannot be used for over-the-air attacks. Evaluators should execute these exploits only within the provided testbed and should not attempt to adapt or deploy them against production networks or devices.

Network Isolation: All experiments run in a self-contained 5G SA simulation environment. No real cellular networks or commercial devices are involved during artifact evaluation.

Data Collection: The artifact does not collect, transmit, or store any personally identifiable information from evaluators. All experimental outputs remain local to the execution environment.

A.2.2 How to access

The artifact is permanently archived on Zenodo at: https://zenodo.org/doi/10.5281/zenodo.17984911

The Zenodo archive serves as the stable, versioned record of the artifact for long-term availability. It contains all components described in the Open Science section of the paper:

- **A1. Simulation testbed.** This component includes both the source-level modifications to the OAI gNB (based on the 2025.w5 branch) and a pre-built Docker image that encapsulates the complete simulation environment with all required dependencies pre-installed.
- **A2. Constraint-driven toolchain.** This component provides the full analysis and test generation pipeline, including specification preprocessing scripts, field dependency extraction, DSL-based mutation logic, and ASN.1 decoding/encoding utilities.
- **A3. Test cases and proof-of-concept exploits.** This

component contains the generated test cases and corresponding proof-of-concept exploit code that operate through the OAI gNB interception path in the simulation environment.

For Phase-2 functionality and reproducibility assessment, we additionally provide a public GitHub repository as a convenient working copy of the artifact: https://github.com/qiqingh/contest_AE_final

The GitHub repository mirrors the content of the Zenodo archive and provides updated documentation and execution instructions to facilitate artifact evaluation. It does not replace the archived Zenodo version.

For the A1 simulation testbed, although the original modified source code is provided for completeness, we ***strongly recommend*** that evaluators use the pre-configured remote environment. This ensures a consistent and stable evaluation environment.

Optional remote access. For reviewer convenience, we additionally provide optional access to a pre-configured remote environment in which the Docker-based A1 and A3 setup is already deployed and ready to run. This option allows evaluators to validate the simulation-based experiments without any local installation effort. Remote access is provided purely as a convenience mechanism and is not required for artifact evaluation. Access details are described in Experiment E2 below.

A.2.3 Hardware dependencies

None. The artifact does not require any specialized 5G wireless hardware. All evaluations can be completed entirely in a simulation environment. In particular, the artifact does not require SDR devices (e.g., USRP), commercial smartphones, SIM cards, antennas, or any over-the-air radio infrastructure.

The artifact runs on a standard x86-64 machine (e.g., Intel or AMD CPUs) and does not require GPU acceleration. A machine with 32 GB of RAM and 80 GB of available disk space is recommended. The full T3 inter-IE pipeline requires more RAM due to peak memory usage during multi-process serialization; evaluators with less RAM may use the lightweight verification alternatively. The artifact has been verified on the following configuration:

- CPU: 12th Gen Intel Core i7-1260P (12 cores)
- RAM: 32 GB
- OS: Ubuntu 22.04 LTS

For reviewer convenience, we additionally provide direct SSH access to a pre-configured remote environment where the Docker-based simulation environment is already deployed and ready to run. Access details are provided in Section A.3.1.

Although the artifact may trigger crashes or assertion failures of the simulated UE or gNB, such behavior is strictly confined to an isolated simulation environment and does not affect real devices or production networks.

A.2.4 Software dependencies

The artifact has been tested on Ubuntu 22.04 LTS and is expected to run on a standard Linux environment. No other operating systems are officially supported for artifact evaluation.

Docker is optional and is provided as a convenience to validate the A1 simulation testbed and A3 test cases. All required system libraries and runtime dependencies for these components are pre-installed in the provided Docker image, and reviewers are not required to compile or install additional third-party software when using this option. Alternatively, reviewers may access a pre-configured remote desktop environment that hosts the same Docker-based setup.

The A2 constraint-driven toolchain runs natively on Ubuntu 22.04 using Python 3.10.12. All Python dependencies are open-source and can be installed via `pip3`. For transparency and reproducibility, we provide an `installed_packages.txt` file generated via `pip3 freeze`, which enumerates the exact Python package versions used in our experiments.

The artifact does not rely on any proprietary or licensed software, vendor-specific SDKs, firmware images, or restricted datasets. Large language models are not invoked during Phase-2 artifact evaluation; all LLM-assisted processing has been performed offline prior to evaluation.

A.2.5 Benchmarks

This work does not rely on traditional machine-learning benchmarks or pre-collected datasets. Instead, the experiments are driven by protocol specifications, reference protocol messages, and target UE implementations.

Specifically, the artifact relies on the following inputs:

- **3GPP protocol specifications.** The constraint-driven analysis (A2) uses publicly available 3GPP technical specifications (e.g., TS 38.331 for RRC) as normative input to extract field-level and cross-field semantic constraints. Relevant specification excerpts used during evaluation are documented and referenced in the artifact.
- **Reference RRC messages.** The test generation pipeline operates on syntactically valid RRC messages that serve as seeds for decode–modify–re-encode mutation. These reference messages are either generated by the simulation environment or included in the artifact as encoded inputs.
- **UE implementations as workloads.** The evaluated workloads consist of the OpenAirInterface UE running in a simulation environment. The UE implementation

serves as the target system under test rather than a benchmark dataset.

All inputs required for the experiments are either publicly available (3GPP specifications) or included in the artifact and do not require any external datasets.

A.3 Set-up

This section describes the installation steps required to prepare the environment for evaluating the CONSET artifact. The artifact follows a two-stage workflow: **E1** runs on the host environment (A2), and **E2** runs inside a Docker-based simulation environment (A1/A3).

A.3.1 Installation

Operating system. The artifact is evaluated on Ubuntu 22.04 LTS (x86-64). No proprietary software is required.

E1 — Python environment (A2). The constraint-driven toolchain (A2) requires Python 3.10 on Ubuntu 22.04. Evaluators may start from a clean Docker container as the host environment:

```
docker run -it --name ae_e1_test ubuntu:22.04 bash
```

Then install the required system packages:

```
apt update && apt install -y \
  git python3 python3-venv python3-pip python3-dev \
  gcc g++ build-essential make cmake pkg-config \
  libcairo2-dev libsystemd-dev gettext \
  libdbus-glib-1-dev libgirepository1.0-dev \
  libdbus-1-dev libcups2-dev
```

Clone the repository and set up a Python virtual environment:

```
git clone https://github.com/qiqingh/contest_AE_final
cd contest_AE_final/A2_constraintdriven_toolchain
python3 -m venv .venv
source .venv/bin/activate
pip3 install -r installed_packages.txt
```

This installs all non-proprietary packages required by the analysis, constraint extraction, and ASN.1 decoding/encoding pipeline.

E2 — Simulation environment (A1/A3). For the simulation-based testbed and proof-of-concept exploits (A1/A3), evaluators may set up the environment locally by pulling the pre-built Docker image:

```
sudo docker pull kqing0515/oai_testing:v3
sudo docker run -it \
  --name oai25_testing \
  --cpus="8" \
  --privileged \
  --ipc=host \
  --network=host \
  --mount type=tmpfs,destination=/dev/shm \
  --mount type=tmpfs,destination=/dev/mqueue \
  kqing0515/oai_testing:v3
```

A.4 Evaluation Workflow

This section describes the evaluation workflow used by the Artifact Evaluation Committee (AEC) to assess the functionality and reproducibility of the CONSET artifact. The workflow focuses on validating the key claims of the paper that are directly supported by the released artifacts, without requiring over-the-air (OTA) experiments on commercial devices.

A.4.1 Major Claims

(C1): *CONSET enables systematic generation of syntactically valid but semantically inconsistent 5G RRC messages by extracting specification-level constraints and cross-field dependencies. This claim is supported by the constraint extraction and DSL-based test generation pipeline described in Sections §4 and §5 of the paper.*

(C2): *Constraint-guided semantic testing with CONSET uncovers real implementation flaws in 5G UE implementations that are difficult to expose using syntax-level or schema-valid mutation alone. This claim is supported by the simulation-based evaluation results reported in Sections §6.2–§6.4 of the paper.*

A.4.2 Experiments

(E1): *Constraint extraction and semantic test generation [30 human-minutes + ∼2.5 compute-hours for full T3 inter-IE (or <10 minutes using the lightweight alternative). Total estimated time: ∼3 hours for full run, or <30 minutes using the lightweight alternative].* This experiment validates Claim (C1) by reproducing the constraint-driven analysis and test generation pipeline of CONSET.

Preparation: Set up the A2 Python environment on the host as described in the Set-up section. Note that **T1** (3GPP PDF preprocessing, requires Mathpix API key) and **T4** (LLM-based DSL synthesis, requires OpenAI API key) are **optional**. Pre-generated intermediate outputs for all stages are included in the repository and can be used to skip these steps.

Execution: All commands run from `contest_AE_final/A2_constraintdriven_toolchain/`. The recommended path is T2 → T3 → T5 → T6 (no API keys required):

T2 — IE Collection:

```
cd toolchain2_IE_collection/intra-IE/code
python3 00_extract_IE_id.py
python3 01_filter_IE_with_ASN.py
python3 02_greedy_set_cover_intra.py && cd -

cd toolchain2_IE_collection/inter-IE/code
python3 00_extract_IE_id.py
python3 01_filter_IE_with_ASN.py
python3 02_greedy_set_cover_inter.py && cd -
```

T3 — Context Extraction (intra-IE):

```
cd toolchain3_field_pair_context_extraction/intra-IE/code
python3 00_intra-IE_context_extract.py --all-pairs --no-self
cd -
```

T3 — Context Extraction (inter-IE, ∼2.5 hours): Verified on 12-core i7-1260P, 32 GB RAM, Ubuntu 22.04. Observed runtimes: 2h19m, 2h28m, 2h23m, 2h29m across four runs.

```
cd toolchain3_field_pair_context_extraction/inter-IE/code
python3 00_generate_aggressive_inter_ie_config.py
python3 01_inter_ie_enhanced_extractor.py && cd -
```

T5 — Test Case Generation:

```
cd toolchain5_dsl_to_testcase/code
python3 unified_test_generator.py && cd -
```

T6 — OTA Payload Generation:

```
cd toolchain6_generate_OTA_testcase/intra-IE/code
python3 ./run_T6.py && cd -

cd toolchain6_generate_OTA_testcase/inter-IE/code
python3 ./run_T6.py && cd -
```

If the full T3 inter-IE run is not feasible (e.g., insufficient RAM), a lightweight alternative is provided that uses the same extractor code with a reduced cluster configuration. Estimated runtime: <10 minutes:

```
cd toolchain3_field_pair_context_extraction/inter-IE/code
python3 01_inter_ie_enhanced_extractor_lite_ae.py
cd -
```

Results: The expected outputs are as follows:

- **DSL rules:** Search for `"has_valid_rule": true` in:
  `toolchain4_field_pair_LLM_query/intra-IE/outputs/intra-IE_DSL_results_gpt4o`
  `toolchain4_field_pair_LLM_query/inter-IE/output/inter_ie_dsl_rules_gpt4o`
- **Generated payloads:**
  `toolchain6_generate_OTA_testcase/intra-IE/output/06_payloads`
  `toolchain6_generate_OTA_testcase/inter-IE/output/06_payloads`

Each payload file contains **deterministic offset-based modifications** derived from the extracted semantic constraints.

(E2): *Simulation-based vulnerability reproduction on OAI UE [30 human-minutes].* This experiment validates Claim (C2) by reproducing crash-triggering behaviors on the open-source OAI UE in a controlled simulation environment.

Preparation: Set up the simulation environment locally as described in the Set-up section. Then enter the Docker container:

```
docker exec -it oai25_testing bash
cd /home/5ghoul-5g-nr-attacks
```

Execution: Step 1 — Baseline run (no exploit):

```
sudo bin/5g_fuzzer \
  --EnableSimulator=true \
  --EnableMutation=false \
  --GlobalTimeout=false
```

Expected: repeated `[!] UE process stopped` / `[!] UE process started` cycles without frequent crashes.

Step 2 — Exploit run:

```
sudo ./bin/5g_fuzzer \
  --exploit=mac_sch_6e99a9e9 \
  --MCC=001 --MNC=01 \
  --GlobalTimeout=false \
  --EnableMutation=false
```

Expected: frequent `[!] UE process crashed` output.

E1→E2 end-to-end validation: To validate that E1 outputs can be directly used in E2, copy any payload generated by E1 (T6) into the Docker container, compile, and run. The following uses `mac_sch_f108_f117` as an example (verified to reliably trigger crashes). If using the provided SSH environment, the repository is located at `/home/qiqingh/Desktop/contest_AE_final`; run the following from that directory:

```
docker cp A2_constraintdriven_toolchain/\
toolchain6_generate_OTA_testcase/inter-IE/\
output/06_payloads/mac_sch_f108_f117.cpp \
oai25_testing:/home/5ghoul-5g-nr-attacks/\
modules/exploits/5gnr_gnb/
```

Then enter the container, compile, and run:

```
docker exec -it oai25_testing bash
cd /home/5ghoul-5g-nr-attacks
sudo bin/5g_fuzzer --list-exploits
sudo ./bin/5g_fuzzer \
  --exploit=mac_sch_f108_f117 \
  --MCC=001 --MNC=01 \
  --GlobalTimeout=false \
  --EnableMutation=false
```

The following E1-generated payloads have been verified to reliably trigger UE crashes, together with their corresponding pre-compiled E2 names available under `A3_test_case_exploits/compiled_payloads/`:

E1 payload name	E2 exploit name
mac_sch_f108_f117	mac_sch_e67cf618
mac_sch_f133_f359	mac_sch_aca1c1fc
mac_sch_f452_f453	mac_sch_90ea1170
mac_sch_f690_f692	mac_sch_1a93ba9a
mac_sch_f694_f697	mac_sch_57890405
mac_sch_f823_f830	mac_sch_a243cd52
mac_sch_f825_f830	mac_sch_91c06e74
mac_sch_f831_f832	mac_sch_a86ecf35
mac_sch_f927_f933	mac_sch_b3a4a80b

Automated E1–E2 End-to-End Validation. To validate that E1-generated payloads can be directly replayed in E2, we provide an au-

tomated script `send_exploit_test.py` under `A3_test_case_exploits/auto-run/`. The script runs the 9 verified crash-triggering E1 payloads sequentially against the OAI UE in the Docker container and saves the logs for each exploit run.
To run the script in the provided SSH environment:

```
cd contest_AE_final/A3_test_case_exploits/auto-run
sudo python3 send_exploit_test.py
```

Note: The script is pre-configured for the provided SSH environment. If running in a different environment, update the following two variables at the top of `run_test()` before executing:

- `container_id`: the Docker container ID or name of the simulation environment (e.g., `oai25_testing`)
- `directory_path`: the path to the directory containing the pre-compiled `.so` payload files

Note on benign errors: The following errors may appear during execution and can be safely ignored:

- **`credentials.json` not found:** An email alert feature inherited from the upstream 5Ghoul framework fails due to missing Gmail credentials. This does not affect exploit execution. See: https://github.com/asset-group/5ghoul-5g-nr-attacks/issues/29
- **`idemptables` errors:** Legacy `iptables` setup scripts from the 5Ghoul environment. These do not affect simulation or exploit behavior.
- **Compiler warnings/errors during exploit compilation:** Running `sudo bin/5g_fuzzer -list-exploits` may output compiler warnings or errors. These do not necessarily indicate a failed build. To verify successful compilation, check whether the corresponding `.so` and `.o` files exist at `/home/5ghoul-5g-nr-attacks/modules/exploits/5gnr_gnb/`.

Results: Frequent `[!] UE process crashed` messages confirm that the exploit payload successfully triggers the vulnerability in the simulated OAI UE, consistent with the crash sites reported in Section §6.3 of the paper.

A.5 Notes on Reusability

The CONSET artifact is designed with modularity and low coupling between components, which facilitates reuse beyond the specific evaluation scenarios presented in this paper.

In particular, the constraint-driven toolchain (A2) is largely decoupled from the underlying execution environment and UE implementation. Its specification preprocessing, field-pair extraction, and DSL-based mutation logic operate on decoded protocol structures and can be reused for other RRC messages or extended to additional 3GPP control-plane protocols with minimal structural changes.

Similarly, the simulation-based test execution pipeline (A1 and A3) separates message construction from delivery mechanisms, allowing alternative backends or execution environments to be integrated without modifying the core analysis logic.

While the current artifact focuses on pre-authentication 5G RRC testing in a simulation environment, the modular design enables researchers and practitioners to adapt individual components of CONSET for related protocol testing or specification-driven analysis tasks.

A.6 Post-Submission Updates

The following details have been updated following further communication with the vendors after the camera-ready submission.

- **Table 1 (CVE assignments):** CVE-2025-20703 corresponds to the csi-MeasConfig entry and CVE-2025-20666 corresponds to the ServingCellConfig entry.
- **Table 1 (Affected IE):** The Affected IE for the third MediaTek row has been updated to ServingCellConfig.
- **Section 6.2:** The number of tested commercial smartphones has been updated to eight.

A.7 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Principled Design of Indexing Functions for Memory Coloring

Stephan Dübler[1], Jana Hofmann[1], Boris Köpf[2], Stavros Volos[2]

[1]Max Planck Institute for Security and Privacy (MPI-SP)
[2]Azure Research, Microsoft

A Artifact Appendix

A.1 Abstract

Our work presents algorithms for synthesizing indexing functions that support memory coloring for partitioning of microarchitectural resources. This artifact contains the code and data for our case studies (presented in Section 7 of the paper).

In the paper, we conduct three experiments: 1) an ablation study in which we synthesize functions for increasingly complicated demands 2) an experiment in which we synthesize functions for a realistic 16-core server-class CPU and 3) a security study, in which we confirm the isolation guarantees provided by our approach.

At the top level, the experimental pipeline proceeds in two stages: 1) synthesize an indexing function from a set of address traces representing accesses to the L3 cache and 2) evaluate the performance of the synthesized function. Both the collection of L3 access traces and evaluation of performance is carried out in ChampSim, a trace-based simulator of the memory hierarchy.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact is provided as a docker container. Running it does not require any particular system setup. The only risk is that a full-fetched simulation will require significant resources for several days.

A.2.2 How to access

The artifact is available on Zenodo: https://doi.org/10.5281/zenodo.18494189. The artifact contains:

- The implementation of our algorithms and various configuration files in `function_synthesis_algorithm.tar`
- Our adapted version of the ChampSim simulator in `Champsim.tar`, which also includes our benchmarks (i.e., workload traces) in `traces`.
- A folder `scripts` with various scripts for function generation and evaluation.
- All synthesized functions produced for the paper (`synthesized_functions_paper.txt` in `synthesized_functions`) and the corresponding full evaluation metrics (`Camera-Ready-Numbers.xlsx`, `Camera-Ready-Numbers.md`)
- A Docker image (Dockerfile , docker-compose.yml)
- A `README.md` file with further information.

A.2.3 Hardware dependencies

Fully reproducing the results provided in Section 7 requires significant compute as we simulate the full memory hierarchy for several millions of instructions. By default, our scripts run 8 such simulations in parallel. We recommend a system with at least 10 CPU cores, 20GiB RAM and 60GiB free disk space for the Docker container to run. If these requirements cannot be met, the number of parallel simulations can be configured (see Section A.5). On Linux, the Docker container's available RAM scales automatically with the system. On macOS, it needs to be configured, e.g., via Docker Desktop.

A.2.4 Software dependencies

The artifact is provided as a docker container and uses docker compose. It has been tested on macOS 26.5. For instructions on how to install docker and docker compose see https://docs.docker.com/compose/install/. When run outside the docker container, the artifact requires `Python 3.11` with the following packages: mpmath 1.3.0, numpy 2.3.2, openpyxl 3.1.5, psutil 7.2.1, PyYAML 6.0.2, scipy 1.16.2, sortedcontainers 2.4.0, sympy 1.14.0

A.2.5 Benchmarks

All workload traces used in the artifact can be found in `Champsim.tar/traces`. These are SPEC2017 and PARSEC ChampSim traces. Our pipeline first generates the L3 access

traces from these workloads, which are then aggregated into a probability distribution.

A.3 Set-up

A.3.1 Installation

We recommend using the Docker image for evaluation. Install Docker and Docker Compose (see above). At the top-level of the artifact run:

```
# 0) Unpack all 4 tar archives
# 1) Build the image
docker compose build
# 2) Start the container
docker compose up -d
# 3) Open a shell inside
docker compose exec artifact bash
```

To generate the necessary training data for all following scripts, either run the full-length script `scripts/generate_training_data.sh` (approx. 12-14 hours) or the shortened variant `scripts/generate_training_data.sh --reduce-sim-instr` (approx. 3-4 hours). To closely reproduce the paper's experiments, we recommend running the full-length script.

A.3.2 Basic Test

In the Docker shell, run `scripts/full_pipeline_basic_test.sh`. This briefly starts all components of our pipeline (generation of probability distribution, function synthesis, and function evaluation). It should not take longer than 1 minute.

A.4 Evaluation workflow

The goal of this artifact is to reproduce the results of **EXP1 - EXP6** (which make claims about the performance of the synthesized functions) as well as the security evaluation (**EXP7** - Section 7.4) of the paper.

A.4.1 Major Claims

The major claims are:

Claim 1 Without coloring requirements, the synthesized functions achieve a performance that is on par with the default L3 set indexing function (within 0.5% difference for hit rate, miss latency and IPC).

Claim 2 With additional coloring requirements, the hit rate gradually decreases.

Claim 3 For a 16-core server class CPU, we can synthesize an L3 function that enables simultaneously partitioning both the L3 cache and DRAM. When enforcing 16 colors, the IPC when using the synthesize functions is reduced by less than 3%.

Claim 4 The coloring scheme obtained from the synthesized functions prevents cross-color L3 cache and row buffer evictions.

A.4.2 Experiments

We propose experiments **EXP1** - **EXP7** mirroring those described in the paper. Each experiment consists of several batches, where in each batch, the algorithm synthesizes a function from randomly selected workloads and evaluates it on different randomly selected workloads.

For the human, each experiment consists of three steps: 1) run a script, 2) pull the results from the Docker container, and 3) inspect the results in `Newly-Calculated_Numbers.xlsx` and check if they match the results given in the paper for the respective experiment.

Every script invokes a chain of scripts that executes all parts of the pipeline. Further information can be found in `README.md`.

Faithfully reproducing all experiments at the scale described in the paper takes at least 9 days if the Docker container has the resources described in Section A.2.3. For each experiment, we therefore provide a full version and a shortened version. The shortened version runs the experiment with fewer simulated instructions (detailed below).

By default, the results are inside the Docker container. To inspect them from host, either first run `exit` from the same terminal and then `scripts/extract_test_results.sh`, or open a separate terminal and run `scripts/extract_test_results.sh` directly.

Runtime estimates calibrated against measured runs on a 12-core Apple M2 Pro inside Docker Desktop with 20 GiB of allocated memory with 8 jobs in parallel.

EXP1 - supporting Claim 1 *[5 human-minutes + 11-13 compute-hours]*: Run `scripts/run_full_pipeline.sh --exp1` (approx. 11-13 hours) or `scripts/run_full_pipeline.sh --exp1 --reduce-sim-instr` (approx. 3-4 hours). The metrics produced by the synthesized functions should differ less than 0.5% compared to the default L3 function.

EXP2 - supporting Claim 1 *[5 human-minutes + 11-13 compute-hours]*: Run `scripts/run_full_pipeline.sh --exp2` (approx. 11-13 hours) or `scripts/run_full_pipeline.sh --exp2 --reduce-sim-instr` (approx. 3-4 hours). As for **EXP2**, the metrics produced by the synthesized function should differ less than 0.5% compared to the default L3 function.

EXP3 - supporting Claim 2 *[5 human-minutes + 40-50 compute-hours]*: Run `scripts/run_full_pipeline.sh --exp3` (approx. 40-50 hours) or `scripts/run_full_pipeline.sh --exp3 --reduce-sim-instr` (approx. 10-12 hours). The metrics should be within the range reported in Fig. 5 of the paper.

EXP4 - supporting Claim 2 *[5 human-minutes + 25-30*

compute-hours]: Run `scripts/run_full_pipeline.sh --exp4` (approx. 25-30 hours) or `scripts/run_full_pipeline.sh --exp4 --reduce-sim-instr` (approx. 7-9 hours). The metrics should be close to the ones reported in Fig. 6 in the paper.

EXP5 - supporting Claim 2 *[5 human-minutes + 30-35 compute-hours]*: Run `scripts/run_full_pipeline.sh --exp5` (approx. 30-35 hours) or `scripts/run_full_pipeline.sh --exp5 --reduce-sim-instr` (approx. 9-11 hours). The metrics should be close to those reported in Fig. 7 in the paper.be

EXP6 - supporting Claim 3 *[5 human-minutes + 70-75 compute-hours]*: Run `scripts/run_full_pipeline.sh --exp6` (approx. 70-75 hours) or `scripts/run_full_pipeline.sh --exp6 --reduce-sim-instr` (approx. 20-25 hours). The metrics should be close to the ones reported in Fig. 8 in the paper.

EXP7 - supporting Claim 4 *[5 human-minutes + 20-25 compute-hours]*: Run `scripts/run_full_pipeline.sh --exp7` (approx. 20-25 hours) or `scripts/run_full_pipeline.sh --exp7 --reduce-sim-instr` (approx. 6-8 hours). This experiment implements a coloring scheme derived from one of the functions synthesized in **EXP6**. The script records the evictions caused by an attacker process for both L3 and DRAM row buffers with and without coloring enabled. The results support Fig. 9 in the paper.

The exact locations of the results within `Newly-Calculated-Numbers.xlsx` are shown in Table 1. The results are duplicated in the markdown file.

Exp.	Result location in `Newly-Calculated-Numbers.xlsx`
Exp1	`Exp1-5-Single-Core *` sheets(1,2,3) - top sub-table cols B (`standard L3`) + C (`Performance`), rows 2-33.
Exp2	`Exp1-5-Single-Core *` sheets(1,2,3) - prefetcher sub-table cols B + C (rows 38-69).
Exp3	`Exp1-5-Single-Core *` sheets(1,2,3) - top sub-table cols E-P (`4k page Ncolors`), rows 2-33.
Exp4	`Exp1-5-Single-Core *` sheets(1,2,3) - top sub-table cols Q-X (`4kpage_dram_Ncolors`), rows 2-33.
Exp5	`Exp1-5-Single-Core *` sheets(1,2,3) - slicing sub-table cols H (`standard L3`) + I-P (`4kpage_dram_slice_Ncolors`), rows 38-69.
Exp6	`Exp6-Multi-Core` sheet(4) - col B (`standard L3 2mb page`) + cols C onward (`2mbpage_dram_slice_Ncolors`), across the Hit-Rate / IPC / Latency sub-sections.
Exp7	`Exp7-Color-Isolation` sheet(5) - LLC + DRAM eviction matrices.

Table 1: Result location in the xlsx file by experiment.

Number of Simulated Instructions Here, we provide more detailed information on the number of simulated instructions in the full (default) vs the short (`--reduce-sim-instr`) version of the above experiments.

- **EXP1 - EXP5**: Full version: 50M warmup, 200M sim. (for both training and evaluation). Short version: training: 10M warmup, 50M sim.; evaluation: 10M warmup, 20M sim.
- **EXP6**: Full version: training: 50M warmup, 100M sim.; evaluation: 50M warmup, 50M sim. Short version: training: 10M warmup, 25M sim.; evaluation: 10M warmup, 10M sim.
- **EXP7**: Full version: no warmup, 50M sim. Short version: no warmup, 1M sim.

A.5 Notes on Reusability

Pipeline Each experiment script partially invokes `scripts/run_full_pipeline.sh`, which can additionally be run with an `--jobs <i>` flag to set the number of parallel simulations to `i`. When intending to run all experiments at once, this script can be run without additional flags, also compatible with `--reduce-sim-instr`. Further details on the functionality of the provided scripts can be found in `README.md`.

Non-Determinism Our algorithms are not completely deterministic. Hence, the resulting functions and performance numbers may vary slightly. The variance in performance metrics is expected to be negligible.

Synthesized Functions For technical reasons, some of the functions in the provided txt files have extra bits appended that are not from our synthesis algorithm but are added by our scripts. Functions labeled exp5 have a 3-bit slicing function prepended, from the simulated Rocket Lake CPU. Functions labeled exp6 have 5 bits appended to the end, which do not carry meaning in themselves. These bits are overwritten by Champsim by the non-linear slicing function from the simulated Xeon CPU.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: An Integer Overflow Endgame: Compiler and Architecture Support for Default-On Integer Overflow Mitigation

Zheng Zhang
Purdue University

Qi Ling
Purdue University

Kian Kasad
Purdue University

Brendan Ryan Sweeney
UT Austin

Deepak Gupta
Meta Platforms Inc.

Tal Garfinkel
Google

Kazem Taram
Purdue University

A Artifact Appendix

A.1 Abstract

To support the replicability of our results and foster future research into integer overflow mitigation, we have made our complete artifact suite available. It contains the following components: modified LLVM and Rust compiler toolchains with opX support, all benchmarks used to evaluate performance (except SPEC CPU 2017), and scripts to build and orchestrate the evaluation. In addition, a modified version of gem5 is also included, although it is only used for functionality tests and not for performance evaluation.

The artifacts also come with a top-level and Rust-specific `README.md` to explain the setup details in case the automatic script runs into issues.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Except for installing dependencies, the software does not send or receive data over the Internet. All computations are performed locally. The code does not execute any destructive steps. For performance stability, address space layout randomization (ASLR) is temporarily disabled during the evaluation and restored afterwards.

A.2.2 How to access

The artifacts are available on Zenodo[1]. They are also available as a public CloudLab profile[2].

A.2.3 Hardware dependencies

A **bare-metal** x86 machine with at least 32 GB of RAM and 150 GB of free disk space is required to evaluate the artifacts. The CPU must support AVX instructions. The evaluation takes approximately 2 days to complete on a server with a 28-core Intel(R) Xeon(R) Gold 5512U CPU on CloudLab.

Note that evaluating on virtual machines (VMs) might result in severe result noises, so it is highly discouraged. If you have to run on VMs, make sure to run `/opt/overflow/scripts/bench.sh` on host to disable SMT and fix frequency.

In addition, compiling the LLVM and Rust toolchains (the build script uses `-j16` by default) can consume significant memory; if the machine has low RAM, ensure that sufficient swap space is configured to prevent Out-Of-Memory (OOM) compiler crashes.

A.2.4 Software dependencies

The installation script assumes a Debian-based Linux distribution, but can be easily adapted to other distributions. The script installs all dependencies. It has been tested on Ubuntu 22.04 LTS.

A.2.5 Benchmarks

We use the following benchmarks: SPEC CPU 2017, LFI-Bench, SQLite, RocksDB, Redis, NPB-Rust, and Hashbrown. All benchmarks are included in the artifacts except SPEC CPU 2017, which we cannot redistribute due to licensing. Please obtain it separately (version 1.1.9 has been tested for the evaluation).

To install SPEC CPU 2017, you can either place the installer ISO image (`cpu2017-1.1.9.iso`) under `/opt` and run `/opt/overflow/scripts/install_spec.sh`, or manually install SPEC CPU 2017 directly into `/opt/overflow/spec2017`.

[1] https://doi.org/10.5281/zenodo.20317245
[2] https://www.cloudlab.us/p/int-overflow-hw/int-overflow-ae

A.3 Set-up

A.3.1 Installation

If a CloudLab account is available, using the CloudLab profile is more convenient. Once the profile is instantiated as an experiment, everything is set up on the machine. If the default `c6620` node is unavailable, other x86 nodes can also be used.

Otherwise, the artifacts can be installed manually from the Zenodo archive:

```
# requires root
sudo tar xfz overflow.tar.gz -C /opt
sudo chown -R $USER /opt/overflow
```

Regardless of the previous step, install dependencies and build toolchains (estimated 30 minutes):

```
cd /opt/overflow
./install.sh
```

Verify that the installation completed successfully:

```
source "$HOME/.cargo/env"
rustc --version
# expected: rustc 1.86.0-dev
/opt/overflow/llvm/bin/clang -v
# expected: clang version 19.1.0
```

A.3.2 Basic Test

Perform a basic test on the built toolchains to verify their functionality:

```
cd /opt/overflow/test
./basic_test.sh
# expected: Basic Functionality Test Complete!
```

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): In SPEC CPU 2017, opX achieves near-zero-cost overflow checks (0.3% without SIMD and 1.9% with SIMD checks) compared to the 14.9% overhead of UBSan. Software checks with LCI achieve 4.4% (11.7% with SIMD checks) overhead, which is significantly lower than UBSan. The results are presented in Figures 8 and 9.

(C2): In popular libraries and CoreMark, opX achieves a 4% overall overhead compared to UBSan's 41.7%. The software-only LCI version also achieves a 19.6% overhead. The results are shown in Figure 12.

(C3): In the Hashbrown benchmark and NPB-Rust, opX achieves near-zero overhead compared to the 3.5% overhead of Rust's built-in overflow checks. The software-only LCI version also achieves near-zero overhead. The results are shown in Figure 13.

A.4.2 Experiments

(E1): *[Benchmarks] [5 human-minutes + 48 compute-hours + 150 GB disk space]: Run all benchmarks to validate our major claims.*

Preparation: Ensure that the installation is complete and the basic test passes.

Execution: Run the following command in tmux to start the orchestration script:

```
cd /opt/overflow
bash reproduce_all.sh > reproduce.log 2>&1
```

`SKIP_SPEC`, `SKIP_LFI`, `SKIP_DB`, and `SKIP_RUST` can be set to 1 to skip SPEC CPU 2017, popular libraries and CoreMark, databases (SQLite, RocksDB, and Redis), and Rust benchmarks if needed.

Results: The figures are generated under `/opt/overflow/results` as PDF files:

- `fig8.pdf`: SPEC CPU 2017 performance with opX
- `fig9.pdf`: SPEC CPU 2017 performance with LCI
- `fig12.pdf`: Popular libraries and CoreMark performance
- `fig13.pdf`: Hashbrown and NPB-Rust performance
- `fig16.pdf`: Database performance (optional)

Compare these figures with the results in the paper.

A.5 Notes on Reusability

Our LLVM and Rust compiler toolchains are currently experimental. They can be used to compile other C/C++ and Rust programs, but may encounter some back-end lowering errors. However, the compilers remain useful even without hardware-accelerated integer overflow checking. For instance, we show that software-only LCI can still outperform standard LLVM UBSan. By default, overflows are logged to a file instead of being printed to stdout. This behavior can be changed in `ubsan_handlers.cpp`.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Khost: KVM-based Near Native MCU Firmware Rehosting

Chunlin Wang, Yicheng Yang, Yuan Zhang, Haoyu Xiao, Yifan Zhang, Jiarun Dai
Fudan University, China

A Artifact Appendix

A.1 Abstract

Khost is a KVM-based near native rehosting framework for MCU firmware. This artifact describes the process of deploying and evaluating Khost on the ARMv8-A platform.

A.2 Description & Requirements

Khost is developed and evaluated on Ubuntu 22.04 using two ARM-based platforms: (1) a Huawei KunPeng server (TaiShan 2280) with dual 32-core Cortex-A72 processors clocked at up to 2.4 GHz and 256 GB DDR4 RAM, and (2) a Raspberry Pi 4B with a quad-core Cortex-A72 processor clocked at up to 1.8 GHz, up to 8 GB LPDDR4 RAM, and 64 GB flash storage. To reproduce our results, we recommend using the same hardware and system configuration.

A.2.1 Security, privacy, and ethical concerns

On the one hand, Khost relies on KVM hardware features and requires specific kernel capabilities. If these features are not properly configured or used, it may lead to instability or even crashes on the host system. On the other hand, new bugs may be discovered during the evaluation of this artifact. We would appreciate responsible disclosure of any such issues to the maintainers.

A.2.2 How to access

Our research adheres to the principles of open science. All artifacts, including datasets, scripts, the source code of our tool, and patches to existing tools for ARM platform support, are publicly available at a stable reference: https://doi.org/10.5281/zenodo.17976459 . In addition, any subsequent updates, including new features or bug fixes, are maintained at https://github.com/seclab-fudan/Khost.

A.2.3 Hardware dependencies

Khost is a KVM-based MCU firmware rehosting framework that requires specific hardware features for reproducibility. First, it relies on ARM platforms that support the ARM KVM extension. Second, it leverages the backward compatibility of the ARMv8-A instruction set architecture, which ensures that most MCU instructions can be executed correctly. Finally, sufficient computational resources are required to reproduce our experiments smoothly, specifically at least 8 GB of RAM and 64 GB of storage.

Currently, Khost has been successfully evaluated on both a Huawei KunPeng server (TaiShan 2280) and a Raspberry Pi 4B development board (8 GB RAM and 64 GB flash storage). Since the experiments reported in this paper were primarily conducted on the KunPeng server, which is located within an internal network and is not publicly accessible due to privacy constraints, we recommend using a Raspberry Pi 4B for artifact evaluation. Although the results obtained on the Raspberry Pi 4B do not match the performance achieved on the Kunpeng server, they are sufficient to reproduce the experiments and demonstrate the key outcomes of our work.

A.2.4 Software dependencies

Khost is developed and evaluated on Ubuntu 22.04 in our experiments. Due to a Linux kernel bug that prevents vCPU state restoration in AArch32 system mode, the kernel must be updated to version 5.15.0.1070 or later to include the corresponding fix. In addition, Khost relies on several third-party tools and libraries, including Capstone, Keystone, and Fuzzware. To deploy Khost, the required dependencies should be installed by following the instructions in Section A.3.

A.2.5 Benchmarks

We evaluate Khost on three different datasets. (1) `CoreMark-PRO`, an industry-standard benchmark designed to measure system performance on complex computational workloads. (2) `Simbench`, a comprehensive microbenchmark suite for evaluating system-level performance of full-system emulators. (3) `Real-world Firmware Dataset`, a newly constructed dataset derived from firmware used in prior work. The prebuilt benchmark cases and firmware binaries are available at experiments.

A.3 Set-up

This section describes the process of deploying Khost on an ARM platform, using the Raspberry Pi 4B as a reference. In total, the environment setup and subsequent installation take approximately two hours.

Memory and Swap Requirements: Khost is memory-intensive, so 8 GB of RAM is recommended. If the available physical memory is insufficient, swap space must be configured to prevent the program from being terminated by the Linux Out-Of-Memory (OOM) killer, although this will degrade performance due to increased memory access latency. The following commands can be used to configure swap space.

```
sudo swapoff /swapfile
sudo fallocate -l 8G /swapfile
sudo chmod 600 /swapfile
sudo mkswap /swapfile
sudo swapon /swapfile
```

KVM Permissions: Access to /dev/kvm is restricted to users in the KVM group. If you encounter the error permission denied accessing /dev/kvm, add your user to the kvm group using `sudo usermod -aG kvm $USER`. To take effect, you must fully log out and log back in after running this command. You can verify the change using `groups | grep kvm`. If /dev/kvm is still inaccessible after re-login, ensure that your kernel has KVM support enabled. **Alternatively**, if the system is accessed by logging in as the root user, the above KVM permission configuration steps are not required.

Patch Kernel Bug: Due to a bug in the Linux kernel that prevents vCPU restoration in AArch32 system mode, the kernel must be updated to version 5.15.0.1070 or later to incorporate the corresponding fix. More details can be found in the following upstream KVM patch.

On Raspberry Pi 4B, you can either run `sudo apt install -y linux-image-raspi && sudo reboot` or download the kernel from linux-kernel-for Raspberry and update manually.

The packages and third-party tools required by Khost can be installed using the following terminal commands.

Listing 1: Packages

```
sudo apt install curl vim cmake make bc git gcc g++ \
 llvm-dev libyaml-cpp-dev tmux ninja-build meson \
 libglib2.0-dev gcc-arm-none-eabi python3-pip \
 python3-virtualenv python3-virtualenvwrapper
```

Listing 2: Keystone

```
git clone https://github.com/keystone-engine/keystone.git
cd keystone && git checkout fb92f32391c6cced8
mkdir build && cd build && ../make-share.sh
sudo make install && sudo ldconfig && cd ../../
```

Listing 3: Capstone

```
git clone https://github.com/capstone-engine/capstone.git
cd capstone && git checkout 106f7d3b949f5dd8e6c
./cmake.sh && make -j && sudo make install && sudo
     ldconfig && cd ../
```

Listing 4: Cargo and Rustup

```
sudo apt install iproute2 cargo
sudo curl --proto '=https' --tlsv1.2 -sSf \
https://sh.rustup.rs | sh
```

Listing 5: Add the following code to ~/.cargo/config

```
[target.armv7-unknown-linux-gnueabihf]
  linker = "arm-linux-gnueabihf-gcc"
```

A.3.1 Installation

Navigate to the root directory of Khost and run the following commands to compile Khost with different fuzzing backends. Due to differences in their configurations, Khost must be recompiled when switching between fuzzing backends. In addition, in the LibAFL version, backend processes may persist even after the program is forcefully terminated. Therefore, before starting a new round of experiments on your platform, it is necessary to ensure that all running khost-libafl-fuzz backend processes have been terminated.

Listing 6: AFL++ Backend

```
./clean.sh && ./build.sh
```

Listing 7: LibAFL Backend

```
./clean.sh && cargo clean && cargo build --release
```

A.3.2 Basic Test

After the compilation completes, you can verify the build by running the following terminal. Both commands will display their respective help messages.

```
./build/runner/khost -h
./build/runner/khost-debug -h
./build/fuzzer/khost-afl-fuzz -h
```

A.4 Evaluation workflow

To evaluate Khost, we conduct three major experiments. You can reproduce them with the following commands (all operations are performed in the root directory of the Khost project).

Listing 8: Configure QEMU

```
wget https://download.qemu.org/qemu-8.2.10.tar.bz2
tar xvf qemu-8.2.10.tar.bz2
cp experiments/patches/qemu/stm32f405_soc.h \
     qemu-8.2.10/include/hw/arm/
cd qemu-8.2.10 && mkdir build && cd build
../configure --target-list="arm-softmmu" --disable-capstone
make -j$(($(nproc) - 1))
cd ../../
```

Listing 9: Performance Test on CoreMark-PRO[1]

```
./scripts/coremark_test_qemu.sh
./scripts/coremark_test_khost.py
./scripts/coremark_test_native.sh
```

[1] When executed, the script interactively prompts the user to choose whether to perform ablation experiments.

Listing 10: Performance Test on Simbench

```
./scripts/simbench_test_qemu.sh
./scripts/simbench_test_khost.py
```

Listing 11: Bypass the AFL++ Coredump Check

```
echo core | sudo tee /proc/sys/kernel/core_pattern
cd /sys/devices/system/cpu
echo performance | sudo tee cpu*/cpufreq/scaling_governor
cd -
```

Listing 12: Fuzzing Test with AFL++ Backend (Usage + Case)
Usage: ./scripts/fuzz_start.py firmware_id/all[2]

```
./scripts/fuzz_start.py 01_PLC
./scripts/fuzz_start.py all
```

Listing 13: Fuzzing Test with LibAFL Backend (Usage + Case)
Usage: ./scripts/libafl_fuzz_start.py firmware_id/all

```
./scripts/libafl_fuzz_start.py 01_PLC
./scripts/libafl_fuzz_start.py all
```

Listing 14: Coverage Calculation with AFL++ Backend (U. + C.)
Usage: ./scripts/fuzz_replay.py firmware_id

```
./scripts/fuzz_replay.py 01_PLC
```

Listing 15: Coverage Calculation with LibAFL Backend (U.+C.)
Usage: ./scripts/libafl_fuzz_replay.py firmware_id

```
./scripts/libafl_fuzz_replay.py 01_PLC
```

Listing 16: Replay a Test Case (Usage + Case)
Usage: ./build/runner/khost-debug firmware_config input_file

```
./build/runner/khost-debug \
experiments/fuzz/01_PLC/config.yml \
experiments/fuzz/01_PLC/crashes/id\:000074\,sig\:00\,src
    \:000414\,time\:9648045\,execs\:5625492\,op\:havoc\,
    rep\:13  \
-l debug -o all
```

Listing 17: Overhead of Coverage Collection

```
./scripts/count_overhead_cov.py
```

A.4.1 Major Claims

In our paper, we make the following four major claims.

(C1): Compared to QEMU, Khost reduces the overhead of handling complex computational tasks in CoreMark-PRO by 90.0% to 95.5% on the Kunpeng server, and even on a Raspberry Pi 4B, it reduces overhead by more than 80%.

(C2): Khost reduces the overhead on system-level tasks of Simbench by up to 55% compared to QEMU on the Kunpeng server, except for the exception test cases, and even on a Raspberry Pi 4B, it reduces overhead by more than 22.5%.

[2]The parameter `firmware_id` specifies a single firmware case (e.g., `01_PLC`), while `all` indicates that all 12 firmware instances are fuzzed simultaneously. Please ensure that your platform has sufficient CPU cores and memory resources when using `all`.

(C3): On the Kunpeng server, Khost achieves up to 197.5× higher throughput and improves basic block coverage by up to 6x compared to the existing fuzzing tool (HALucinator).

(C4): Khost is enable to uncover new bugs.

(C5): The overhead caused by coverage collection is approximately 4.71%.

A.4.2 Experiments

To demonstrate our major claims and open-science contributions, we provide the following experiments (approximately nine days of runtime with sufficient hardware resources):

(E1): [Performance Test with CoreMark-PRO] *[10 human-minutes + 7 compute-hours (three rounds) + 64GB disk]*: Evaluate the performance of Khost in handling complex computational tasks, with expected results close to those described in Claim C1 in Section A.4.1. As discussed in our paper, disabling the Auxiliary Page Table (APT) causes all memory regions to be marked as Device-nGnRnE, which severely degrades rehosting performance and requires more than 5 compute-hours to complete the tests. Without the ablation experiments, all tests can be completed within one compute-hour.
Steps: First, deploy Khost on the target platform according to Section A.3. Then, build QEMU following the commands in Listing 8. Next, start performance tests using the commands shown in Listing 9. Finally, the result of each test case will be printed as a log entry, where QEMU reports the metric as `Time Interval` and Khost reports it as `passing time`. More details and examples can be found at: performace-test-on-coremark-pro

(E2): [Performance Test with Simbench] *[10 human-minutes + 210 compute-hours (three rounds) + 64GB disk]*: Evaluate the performance of Khost in handling system-level tasks, with expected results close to those described in Claim C2 in Section A.4.1.
Steps: After completing the configuration in E1, performance tests on Simbench can be directly launched using the commands shown in Listing 10. As discussed in our paper, disabling the APT or moving rehosting components outside KVM severely degrades rehosting performance, requiring more than 208 compute-hours to complete the tests. Without ablation experiments, all tests can be completed within 3 compute-hours.

(E3-1): [Fuzzing Test with AFL++ Backend] *[5 human-minutes + 120 compute-hours (five rounds with all 12 firmware tested simultaneously) or 1440 compute-hours (five rounds with firmware tested one by one) + 64GB disk]*: Evaluate the throughput and basic block coverage of Khost during fuzzing with the AFL++ backend, with expected results close to those described in Claim C3 in Section A.4.1.
Steps: First, deploy Khost on the target platform accord-

ing to Section A.3. Then, you can conduct fuzzing tests with AFL++ backend using the commands shown in Listing 12. Finally, the results of fuzzing can be calculated with commands in Listing 14.

(E3-2): [Fuzzing Test with LibAFL Backend] *[5 human-minutes + 120 compute-hours (five rounds with all 12 firmware tested simultaneously) or 1440 compute-hours (five rounds with firmware tested one by one) + 64GB disk]*: Evaluate the throughput and basic block coverage of Khost during fuzzing with the LibAFL backend, with expected results close to those described in Claim C3 in Section A.4.1.

Steps: After E3-1, you can start fuzzing tests with LibAFL backend using the commands shown in Listing 13. Finally, the results of fuzzing can be calculated with commands in Listing 15.

(E4): [Uncover Bugs] *[5 human-minutes + 24 compute-hours + 64GB disk]*: Evaluate the ability of Khost in finding bugs, with expected results close to those described in Claim C4 in Section A.4.1.

Steps: You can start fuzzing the firmware to find bugs. Alternatively, to facilitate evaluation, we also upload the crash cases to the repository. The seeds that trigger crashes can be found in the crashes directory under the echo firmware subdirectory, for example, `experiments/fuzz/01_PLC/crashes`. Any crash case can be replayed using the commands shown in Listing 16.

(E5): [Overhead of Coverage Collection] *[5 human-minutes + 3.5 compute-hours (3 rounds) + 64GB disk]*: Evaluate the overhead of rewrite-based coverage collection in Khost, with expected results close to those described in Claim C5 in Section A.4.1.

Steps: After deploying Khost, run the test by executing the commands shown in Listing 17.

A.5 Notes on Reusability

This document primarily describes the workflow for using Khost. Detailed instructions on deploying Fuzzware and HALucinator are available at Run Fuzzware on ARM64 and Run HALucinator on ARM64, respectively.

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/usenixsec2026/`.



USENIX Security '26 Artifact Appendix: When HTTP 402 Meets the Blockchain: Risks on Emerging x402 Payments

Qinying Wang*
EPFL

Yong Yang*
Zhejiang University

Yuan Chen
Independent Researcher

Shouling Ji✉
Zhejiang University

Mathias Payer
EPFL

A Artifact Appendix

A.1 Abstract

x402 is an emerging payment protocol for Web APIs and autonomous AI agents. It extends HTTP 402 with blockchain-based payment proofs and delegates verification and settlement to third-party facilitators. In our paper, we present the first systematic security study of real-world x402 facilitator deployments. We identify security rules for authorization correctness and execution safety, build a semi-automated black-box testing tool called X402SCOPE, and evaluate 15 major facilitators. Our study uncovers widespread violations that can lead to free shopping, asset theft, service denial, and gas abuse. We further conduct a large-scale on-chain measurement of 119 million x402-related transactions on Base and Solana to quantify adoption, facilitator centralization, settlement failures, sponsor-paid costs, and on-chain risk signals.

This artifact appendix describes our artifact for x402scope. The artifact contains the source code and documentation for (i) the x402scope testing framework, including test cases, configuration files, deployment instructions, and original experimental logs and supporting evidence for facilitator evaluations, where available; (ii) a controlled Coinbase testnet setup for validating representative facilitator-testing functionality; and (iii) analysis scripts for reproducing the measurement pipeline reported in the paper.

A.2 Description & Requirements

This section describes the software and hardware requirements to reproduce the experimental setup for X402SCOPE.

A.2.1 Security, privacy, and ethical concerns

The main risk is that facilitator tests may interact with remote services. The original facilitator evaluation in the paper involved controlled interactions with third-party production facilitators. These interactions were performed by the authors with rate limits, bounded test cases, our own accounts and resources, and responsible disclosure to affected parties. However, we do not require artifact evaluators to repeat these production tests. Evaluators should use the controlled Coinbase testnet workflow unless they have explicit authorization from the corresponding service operator. Evaluators should avoid storing private keys in the repository and keep API credentials private. In particular, filled `config.py` files must not be committed or shared. The measurement scripts operate on public on-chain data and do not require private user data.

A.2.2 How to access

The artifact is available through three channels.

- **Public archival artifact:** https://zenodo.org/records/20328961
- **Public source repository:** https://github.com/HexHive/x402scope
- **Restricted full artifact for evaluation:** https://zenodo.org/records/20329070

These Zenodo concept records identify the artifact-evaluation releases, provide access to their complete version histories, and resolve to the latest available version of each artifact. The public artifact contains the sanitized x402scope framework, measurement code, configuration templates, MariaDB/MySQL schema and ingestion scripts, SQL queries/views, and plot/table regeneration scripts. The restricted artifact contains the facilitator-evaluation bundle under `FacilitatorEvaluation/`, including the SR pass/fail matrix, per-facilitator summaries, raw `/verify` and `/settle` HTTP evidence logs, and attack/validation logs, where applicable. We also provide helper contracts used in bounded validation.

A.2.3 Hardware dependencies

No special hardware such as GPUs, FPGAs, trusted execution environments, or hardware performance counters is required. A commodity x86-64 Linux machine or virtual machine is

sufficient for inspecting the artifact and running the security-rule scripts on bounded targets.

For reproducing the full blockchain data ingestion workflow, we recommends at least 1TB of SSD storage, because the workflow downloads and processes historical Base blocks/receipts and Solana transaction history. The measurement workflow also requires enough memory and CPU resources to run a MariaDB or MySQL database and execute the analysis queries. For lightweight inspection or regenerating figures from already processed tables, substantially less storage is sufficient.

A.2.4 Software dependencies

The artifact targets Linux, with Ubuntu 22.04 or newer recommended. It requires Python 3.10 or newer and a running MariaDB or MySQL instance for the measurement pipeline. Python dependencies are listed in `requirements.txt` and can be installed in a virtual environment. The artifact also requires local configuration through `config.example.py`, which should be copied to a private `config.py` and filled with database settings, RPC endpoints, facilitator credentials where applicable, and test-account keys.

Reproducing the full measurement pipeline requires reliable RPC access to Base and Solana. In practice, we recommend high-throughput or archival RPC endpoints such as Alchemy. Evaluators using free-tier endpoints may need to lower concurrency settings to avoid rate limits and timeouts.

A.2.5 Benchmarks

No standard benchmark suite or pre-packaged benchmark dataset is required. The artifact uses public Base and Solana blockchain data that can be re-collected with the provided scripts.

A.3 Set-up

A.3.1 Installation

First, download and unpack the artifact, then enter the artifact directory (e.g., `cd x402scope`). Create a Python virtual environment and install all required dependencies:

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

Next, create a local configuration file from the template using `cp config.example.py config.py`.

Then edit `config.py` to fill in local runtime parameters. These include database credentials for MariaDB/MySQL, Base and Solana RPC endpoints, and EVM and Solana test-account private keys. If the evaluator runs facilitator checks that require service credentials, the corresponding fields, such as Coinbase `CDP_API_KEY_ID` and `CDP_API_KEY_SECRET`, should also be filled locally. Unused credential fields can be left blank.

For test accounts, evaluators may generate an EVM account with any EVM-compatible wallet, such as Rabby. For Solana, the artifact provides a helper script to generate a test keypair. Before running testnet or devnet experiments, evaluators should fund these accounts with test tokens. Test USDC can be obtained from Circle's faucet at `https://faucet.circle.com/`: select **Base Sepolia** for EVM test USDC used by the Base Sepolia flow, and **Solana Devnet** for Solana Devnet test USDC. If gas is needed, use the Base Sepolia faucet at `https://www.alchemy.com/faucets/base-sepolia` and the Solana Devnet faucet at `https://faucet.solana.com/`.

For the security-rule verification scripts, copy or link the local configuration file into the `SecurityViolation/` directory:

```
cp config.py SecurityViolation/config.py
```

A.3.2 Basic Test

As a minimal functionality test, evaluators can run the controlled Coinbase Base Sepolia v2 target. After filling `config.py` with testnet credentials and copying it to `SecurityViolation/config.py`, run:

```
python3 SecurityViolation/verify_settle_base_v2.py -t
    coinbase-test
```

The script loads the target metadata, local configuration, RPC endpoint, and test-account keys, and then executes the verify/settle checks. A successful run should complete without an uncaught exception. The correctness test should report HTTP 200 with `isValid: True`, followed by HTTP 200 with `success: True` and a non-empty transaction identifier. The script should then continue to the configured mutation tests and print their outcomes. This Basic Test is a minimal EVM/Base smoke test; the full E1 workflow additionally runs the Solana checks and the server/client, ERC-1271, and ERC-6492 tests.

A.4 Evaluation workflow

This section describes two artifact-evaluation workflows. The first validates that the X402SCOPE security-rule verification framework runs on controlled targets. The second reproduces the empirical measurement pipeline over public Base and Solana on-chain data and regenerates the reported figures and tables.

A.4.1 Major Claims

(C1): X402SCOPE *implements a semi-automated black-box testing framework for checking authorization-correctness and execution-safety rules in x402 facilitators. This claim is supported by the implementation*

in `SecurityViolation/`, the target/configuration files, and the controlled testnet workflow. This is proven by the experiment (E1).

(C2): *The on-chain measurement pipeline reproduces the paper's empirical results on x402 adoption, facilitator concentration, gas and fee consumption, settlement failures, revert reasons, and associated token account creation patterns on Base and Solana. This claim is supported by the scripts in `Measurement/` and `Measurement/paper_figdata/`, together with the MySQL-compatible `x402db` database snapshot provided with the artifact. The reproduction workflow regenerates the measurement CSVs, Figures 3–7, and Tables 4–6 reported in the paper. This is proven by experiment (E2).*

A.4.2 Experiments

(E1): X402SCOPE controlled functionality test [30–60 human-minutes + 10–30 compute-minutes]. This experiment validates C1 by running the X402SCOPE framework on a controlled target. Following the methodology in Section 5.2 of the paper, this experiment exercises the main EVM/Base and Solana rule-checking workflows on the controlled Coinbase testnet and devnet targets.

Preparation: Prepare the environment as follows:
(1) Install the Python dependencies;
(2) Create `config.py` from `config.example.py`;
(3) Fill in Base and Solana testnet accounts, RPC endpoints, and Coinbase CDP credentials, and copy the configuration to `SecurityViolation/config.py`;
(4) Before running any state-changing test, run the read-only preflight check for each selected target:

```
python3 SecurityViolation/preflight_check.py -t
   coinbase-test --check-balances
python3 SecurityViolation/preflight_check.py -t
   coinbase-solanadev --check-balances
```

Note that the target names used below correspond to the controlled Coinbase Base testnet and Solana devnet entries in `SecurityViolation/target.py`. The artifact includes test implementations for both x402 v1 and v2 payload formats.

Execution: Run the following functionality tests:
(a) **Basic verify/settle checks.** Run the basic rule-checking tests for EVM/Base and Solana. These scripts exercise the verify/settle workflow with mutated payloads on the controlled targets. For ordinary verify/settle tests, evaluators should set `valid_before_offset` to greater than 180 seconds.

```
python3 SecurityViolation/verify_settle_base_v2.py
    -t coinbase-test
python3 SecurityViolation/verify_settle_solanav2.py
    -t coinbase-solanadev
```

(b) **EVM server/client flow.** Run the server in Terminal 1 and keep it running:

```
python3 SecurityViolation/server_app_v2.py -t
   coinbase-test
```

Then run the client in Terminal 2:

```
python3 SecurityViolation/evmclient_v2.py -t
   coinbase-test
```

Note that evaluators may adjust `valid_before_offset` in the target/configuration file to check validity-window thresholds (e.g., 8).
(c) **ERC-1271 test.** With the server from Step (b) still running, execute the ERC-1271 client test from Terminal 2:

```
python3 SecurityViolation/erc1271testv2.py -t
    coinbase-test
```

(d) **ERC-6492 test.** Run the ERC-6492 support test. The script includes two variants: the default asset-theft variant and a child-contract deployment variant. Evaluators can find the variant selection instructions in the code comments of `erc6492testv2.py` and run the variants separately.

```
python3 SecurityViolation/erc6492testv2.py -t
    coinbase-test
```

Results: Interpret all the results of the four functionality tests as follows:
(a) **Basic EVM/Base and Solana verify/settle checks.** For both the EVM/Base and Solana scripts, confirm the following:
(i) The script completes without an uncaught exception or implementation error.
(ii) The `Correctness Test` completes successfully. The `/verify` request should return `HTTP 200` and `isValid: True`, while the `/settle` request should return `HTTP 200` and `success: True`. The output should also contain a non-empty transaction identifier.
(iii) After the correctness test, the script continues to execute the configured mutation tests, such as `scheme_test`, `network_test`, `pay_amount_test`, and `insufficient_balance_testing`, and prints their verification and settlement outcomes.

A successful EVM/Base and Solana correctness test produces output similar to:

```
############ Correctness Test
HTTP 200
isValid: True
HTTP 200
success: True
transaction: 0x...
```

The subsequent mutation tests should also be executed and report their outcomes, for example:

```
############ scheme_test
HTTP 400
isValid: False
success: False

############ pay_amount_test
HTTP 400
isValid: False
reason: amount_too_low
HTTP 400
success: False
```

Note that the exact rejection reason may vary with the current Coinbase deployment. These mutation-test outcomes are diagnostic and are not used as the functionality pass criterion.

(b) **EVM server/client flow.** Keep the server running in Terminal 1 and execute the client in Terminal 2. Confirm the following:

(i) The server starts successfully and listens on `http://127.0.0.1:8001` without an uncaught exception. A successful server startup produces output similar to:

```
Serving Flask app 'server_app_v2'
Running on http://127.0.0.1:8001
Press CTRL+C to quit
```

(ii) The client completes without an uncaught exception and prints the sender address, account balance, and generated payment header.

A successful client run therefore produces output similar to:

```
MYADDR: 0x...
MY Balance: ...
paymentheader:
...
<Response [200]> {
...
'X-Settle-Status': 'success',
...
}
paid content
```

This step is considered successful when the server remains available, the client exits without an implementation error, the payment request returns `HTTP 200`, the settlement status is `success`, and the protected content is returned.

(c) **ERC-1271 test.** With the server from Step (b) still running, execute the ERC-1271 client in Terminal 2. Confirm that the client completes without an uncaught exception and prints the sender address, ERC-1271 contract address, HTTP response, and settlement status.

A representative output is:

```
MYADDR: 0x...
CONTRACT: 0x...
<Response [200]>
X-Settle-Status: failed
paid content
```

The supplied server implementation preserves the vulnerable behavior used for the free-shopping proof of concept. Therefore, a response containing `X-Settle-Status: failed` together with `paid content` demonstrates the reported behavior, but this exact outcome is not required to establish artifact functionality. This step is considered successfully executed when the client runs without an implementation error and reports the HTTP and settlement results.

(d) **ERC-6492 test.** Confirm that the script completes without an uncaught exception and prints the `/verify` and `/settle` results. For the current Coinbase deployment, the expected output is:

```
verify response: <Response [400]>
isValid: False

settle response: <Response [400]>
success: False
```

Coinbase deployed an allowlist-based mitigation following our responsible disclosure. Therefore, rejection of the ERC-6492 payload is the expected current behavior and does not indicate an artifact failure.

(E2): On-chain measurement reproduction [10–30 human-minutes + several compute-minutes, depending on database load].

This experiment validates C2 by regenerating the paper's measurement results from the read-only MySQL-compatible `x402db` supplied with the restricted artifact. The supplied `x402db` is the official processed-data snapshot for the Results Reproduced evaluation. It contains the processed Base and Solana records, derived tables, and database views required by the reproduction scripts. Full raw-chain re-ingestion is not required for artifact evaluation. The raw-data ingestion and processing scripts under `Measurement/` are provided only as an optional workflow for reuse, extension, and independent re-collection of later Base and Solana chain data.

Preparation: Install the Python dependencies as described in Section A.3.1. Then enter the measurement reproduction directory and create the local database configuration file:

```
cd Measurement/paper_figdata/
cp config.example.py config.py
chmod 600 config.py
```

Fill `config.py` with the read-only database credentials supplied with the restricted artifact. A local MariaDB/MySQL instance, Base or Solana RPC endpoints, and blockchain test accounts are not required for this experiment.

Evaluators may verify the database connection with:

```
PYTHONPATH=.. python3 - <<'PY'
from runsql import runsql
print(runsql("SELECT 1;"))
print(runsql("SELECT DATABASE();"))
```

```
PY
```

The expected output is:

```
[(1,)]
[('x402db',)]
```

Execution: Generate the intermediate measurement CSVs:

```
python3 transaction_activity.py
python3 daily_gas_fee_trend.py
python3 gas_consumption.py
python3 revert_statistics.py
python3 top_facilitators.py
python3 ATA_rent_events.py
python3 ATA_owner_distribution.py
```

Regenerate Figures 3–7:

```
python3 plot/plot_fig3.py
python3 plot/plot_fig4.py
python3 plot/plot_fig5.py
python3 plot/plot_fig6.py
python3 plot/plot_fig7.py
```

Regenerate Tables 4–6:

```
python3 table/table4.py
python3 table/table5.py
python3 table/table6.py
```

Results: Successful execution regenerates the intermediate measurement CSVs documented in `Measurement/paper_figdata/README.md`, together with:

```
plot/fig3.pdf
plot/fig4.pdf
plot/fig5.pdf
plot/fig6.pdf
plot/fig7.pdf
table/table4.csv
table/table5.csv
table/table6.csv
```

The expected result is a set of regenerated measurement CSVs, figure PDFs, and table CSVs corresponding to Figures 3–7 and Tables 4–6 in the paper. These outputs cover transaction activity, contributor trends, facilitator concentration, gas and fee consumption, settlement failures, revert reasons, ATA rent events, and ATA creation patterns on Base and Solana. Minor differences may arise if evaluators re-ingest data at a later chain height or use different RPC providers, but the scripts should reproduce the same measurement methodology.

A.5 Notes on Reusability

Users can reuse X402SCOPE by adding new facilitator entries to `SecurityViolation/target.py` and supplying their own test accounts, RPC endpoints, and credentials through `config.py`. The measurement pipeline can be rerun on later Base and Solana chain data by updating the data collection range and facilitator address lists. The artifact separates runtime secrets from source code through `config.example.py`, making it portable across RPC providers, databases, and test environments.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: KernelRCA: Facilitating Root Cause Analysis of Memory Corruptions in Linux Kernel with Contextual Causality Chain

Kangzheng Gu
Fudan University

Yifan Zhang
Fudan University

Yuan Zhang
Fudan University

Min Yang
Fudan University

A Artifact Appendix

A.1 Abstract

This artifact contains the complete implementation of KernelRCA, an automatic root cause analysis system for Linux kernel memory corruptions. It consists of three components: (1) the system implementation of KernelRCA; (2) the dataset used in the paper's evaluation; and (3) evaluation scripts for measuring the performance of KernelRCA. These components enable the reproduction of the main results reported 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 available on Zenodo at https://zenodo.org/records/18410035. Please use the latest version of this record. Alternatively, the implementation is also available on GitHub at https://github.com/seclab-fudan/KernelRCA and https://github.com/xiaoguai0992/KernelRCA. Note that the GitHub repositories include only the implementation of KernelRCA and the dataset, and do not contain the evaluation scripts used to reproduce the main results reported in the paper.

A.2.3 Hardware dependencies

To run KernelRCA, the underlying CPU must be an Intel platform with KVM available. The artifact requires a machine with a minimum of 16 CPU cores, 32 GB of RAM, and 500 GB of disk storage. Increasing the number of CPU cores can accelerate Linux kernel compilation, thereby reducing the end-to-end execution time of KernelRCA.

A.2.4 Software dependencies

KernelRCA is evaluated on Ubuntu 18.04. We recommend installing the system using the image available at https://releases.ubuntu.com/18.04/ubuntu-18.04.6-desktop-amd64.iso (kernel version 5.4.0-84-generic). Other Ubuntu versions may cause failures in S2E tracing. The `README.md` included in the artifact provides detailed instructions for installing all required software dependencies. An active Internet connection is required during the installation and execution of KernelRCA.

KernelRCA **cannot** be deployed in a Docker environment. If KernelRCA is installed inside a virtual machine, please ensure that KVM nested virtualization is correctly enabled on the host system so that the virtual machine can access hardware-assisted KVM acceleration.

A.2.5 Benchmarks

The dataset is included in the artifact.

A.3 Set-up

A.3.1 Installation

Before installation, we recommend using a clean installation of Ubuntu 18.04. Detailed installation instructions of KernelRCA are provided in the `README.md` included in the artifact.

A.3.2 Basic Test

After installation, navigate to the `KernelRCA/rca` directory and execute the following command:

```
python3 run.py crash_b66d8de2cec1e3878a0524807b93d96bba182fba
```

This command performs root cause analysis on the example vulnerability. Upon completion, the output report is expected to be generated at

```
KernelRCA/s2e/projects/
    crash_b66d8de2cec1e3878a0524807b93d96bba182fba/
    report.txt
```

The file `report.txt` contains the final root cause analysis report produced by KernelRCA.

If an error occurs during the image build process in KernelRCA, a helpful debugging approach is to examine the build logs and identify the first error that appears. A common cause is network-related issues during the Docker image build or the Debian image build either inside or outside the VM. Problems occurring inside the VM can be diagnosed by connecting to the VM through the VNC interface. If users wish to rerun the image build from scratch, they should remove all files under `s2e/images/*`, particularly the `.stamps` and `.tmp-output` directories within this path.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): KernelRCA successfully generates root-cause reports for the 54 vulnerabilities described in Section 4.2.

(C2): KernelRCA outperforms existing techniques, i.e., crash reports and Syzbot Cause Bisection, in terms of the root-cause distance metric defined in Section 4.2. The comparative results are presented in Table 3.

(C3): The overhead statistics of KernelRCA, described in Section 4.3.1 and illustrated in Table 8, show that the average trace size is 1.39 GB and diagnosing a bug takes approximately 97 seconds on average.

A.4.2 Experiments

(E1): *[Effectiveness] [2 human-hour + 24 compute-hour (all cases, about 30 minutes per-case) + 500 GB disk]:*
How to: Install the KernelRCA, perform the root cause analysis for vulnerabilities in the dataset, and review the root-cause report. Since performing root cause analysis on all cases is time-consuming, we recommend sampling a subset of cases to run and using their results to calculate statistics for the remaining evaluations.
Preparation: Build and install KernelRCA following the instructions in `README.md`. Ensure that the basic test produces the expected root cause report `report.txt`.
Execution: (Option 1, recommended) Navigate to the `KernelRCA/rca` directory and execute:

```
python3 run.py <crash_xxxx>
```

This command will automatically analyze the specified vulnerability and generate a root cause report.
(Option 2) Navigate to the `KernelRCA/rca` directory and execute:

```
python3 run_batch.py
```

It will automatically run all the available cases. Note that some cases, such as `4b1e841`, may generate very large traces (hundreds of gigabytes), which can consume significant time and storage, potentially causing the analysis to appear stalled. To skip such cases, you can manually create a folder named `KernelRCA/s2e/projects/<crash_id>` and place an empty `report.txt` file inside it. The system will then detect and skip the designated case.
Results: The results will be saved in `KernelRCA/s2e/projects/<crash_id>/report.txt`. Users need manually review the root cause report and assess its correctness according to Section 4.1 of the paper.

(E2): *[Comparison with Baselines] [5 human-minute + 10 compute-minute (per-case)]:*
How to: Run the scripts to calculate and compare the root-cause distances of KernelRCA and baselines for a given case.
Preparation: Ensure that KernelRCA has been run on the designated case and that the `report.txt` file is available.
Execution: Navigate to the `KernelRCA/eval` directory and run:

```
./eval_distance.sh <crash_id>
```

This command automatically calculates the root-cause distances for KernelRCA's result and the baselines'. The output is saved in `KernelRCA/s2e/projects/<crash_id>/distance` and will look like:

```
kernelrca_distance: 12
syzbot_distance: 123456
crashreport_distance: 12345
```

Note that the reported distances may vary from those in the paper, as execution traces can differ between runs. In most cases, KernelRCA's root-cause distance should be smaller than those of the baselines, supporting claim (C2).

(E3): *[Efficiency] [1 human-minute + 1 compute-minute (per-case)]:*
How to: Run the scripts to calculate the overhead of KernelRCA for a designated case.
Preparation: Ensure that KernelRCA has been executed on the specified case and that the `report.txt` file is available.
Execution: Navigate to the `KernelRCA/eval` directory and run:

```
python3 eval_overhead.py <crash_id>
```

This command automatically calculates the overhead of KernelRCA for the specified case. The output will be similar to:

```
Trace Size: 6.14 (GB)
Total Time: 299 (s)
Tracing Time: 171 (s)
Norm Time: 36 (s)
Context Rebuild Time: 67 (s)
RCA Time: 25 (s)
```

Note that these statistics may vary from those reported in the paper due to differences in execution traces between runs and variations in hardware environments. On average, the results should support claim (C3) when the conditions described in the paper are met (which do not correspond to the minimal environment requirements). If you find that the execution time for a case is significantly longer than reported (e.g., around 30 minutes or more), this is likely because most of the time is spent on kernel compilation. Alternatively, user can run `run.py` with parameters such as `--only rca` to execute only a single analysis step. This execution time should be closer to the numbers reported in the paper.

A.5 Notes on Reusability

The functionality of KernelRCA is modularized to promote reusability. For example, by passing the argument `-only trace` to `run.py`, KernelRCA performs tracing only. The dynamically collected tracing data are organized in Protobuf format, which may be useful for other program analysis tasks such as taint analysis, dynamic program slicing, and more.

To adapt KernelRCA to kernel versions outside the 5.0–5.14 range, users need to extend `rca/patch_kernel.py` to patch the kernel accordingly, enabling S2E instrumentation as well as KernelRCA's instrumentation to capture the required runtime data.

If users wish to apply KernelRCA to additional types of vulnerabilities by adding new detection rules for unexpected program behaviors, they can refer to `rca/analyzer/cpp/src/RootCauseAnalyzer.cc` to implement their own detectors.

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/usenixsec2026/.







USENIX Security '26 Artifact Appendix: Discovery, characterizing and exploiting controllable-copy objects for kernel data-only attacks with CopyKat

Jakob Koschel[†*] Andrea Mambretti[†*] Alessandro Sorniotti[†*] Pietro Moretto[†]
Claudio Migliorelli[†] Andrea Di Dio[‡] Cristiano Giuffrida[‡] Anil Kurmus[†]

[†]IBM Research Europe – Zurich [‡]Vrije Universiteit Amsterdam

[*]These authors contributed equally

A Artifact Appendix

A.1 Abstract

In this work, we build a tool, CopyKat, capable of finding, verifying and characterizing Controllable Copy Objects (CCOs) within the Linux kernel. These objects can be used to amplify exploitation primitives into more powerful ones. For instance, they can allow a limited out-of-bound write on the heap to turn into arbitrary write. In our work, we provide examples of such use through the construction of 8 end-to-end exploits. Having tools like CopyKat can help discovering and categorizing such sensitive objects and can help with constructing principled defenses.

CopyKat is composed of three different stages. The first component is a modified Syzkaller capable of identifying candidate reproducers that follow the *CCO pattern* introduced in our work. This component runs a lightweight taint analysis on top of an instrumented kernel and a modified version of KASAN. The second component performs a more precise taint analysis using PANDA, where we verify the correctness of the taint and remove false positives. The third component takes the verified reproducers, the more precise taint information, and runs the reproducer under concolic execution. This component symbolize three main areas of interest, and perform the simulated corruption of the CCO object. It verifies that the primitive highlighted by the taint is still reachable after corruption, and it provides the path-constraints that an attacker needs to satisfy to use such object as described by that specific reproducer.

For this artifact evaluation, we provide the instructions on how to use and reproduce our taint verification and concolic execution analysis and their results. While we provide the code patches (on Zenodo [1]) to run our fuzzing campaign through Syzkaller, we do not include it in this artifact evaluation. This choice is dictated by the fact that it would not be possible to guarantee that similar results would be found to compare against our paper findings, owing to the stochastic nature of fuzzing. Furthermore, such stage alone would require much more time than allowed to replicate our results. We provide the results of our fuzzing campaign as starting seeds to the two remaining analyses.

[1] https://doi.org/10.5281/zenodo.18500424

A.2 Description & Requirements

Our artifact contains two separate projects. Both are based upon large open-source projects like PANDA and S2E that in turns have several dependencies. To ease deployment, we use Docker to guarantee that such dependencies are easily satisfied.

While our paper proposes a pipeline, we have setup the two projects under evaluation as independent components to ease their inspection. Our taint analysis takes as input the reproducers from the fuzzing campaing and produces as output the object database. Meanwhile, the concolic execution takes as input the already generated PANDA database that we used during our paper evaluation and provides as output the path constraints of the verified objects.

Hereafter, we will indicate the requirements and the instructions for each stage of the pipeline separately.

A.2.1 Security, privacy, and ethical concerns

Our two analyses do not pose any security risk to the evaluator's machine beyond occupying large amount of disk space. The taint verification component requires to run `python` scripts and `gcc` on the evaluator's machine to transform and integrate the database throughout the experiment, and generate the executable reproducers from the `.cprog` files produced by Syzkaller. The PANDA analysis is run within a docker

container with no access to user data beyond our reproducers. Similarly, the concolic execution is also fully packaged within a docker container and none of its components is run directly on the evaluator's machine besides the `docker build` command.

A.2.2 How to access

Our main artifact to reproduce the paper results can be found on public Github at https://github.com/IBM/CopyKat. The repository hosts the code of our two CopyKat analyses and all necessary input data to re-run and obtain the results of our paper. Meanwhile, on Zenodo at https://doi.org/10.5281/zenodo.18500424, we provide all the code under patch format, including also the code for fuzzing campaign, and our own evaluation results the paper is based on.

A.2.3 Hardware dependencies

CopyKat is a tool developed for x86 architecture (Intel/AMD). It produces a large quantity of data and, if scaled to several CPU, it can be memory hungry. While the system would run on a small scale machine (min 8 cores and 16GB of RAM), for performance reasons, we recommend at least 16 cores and 32GB of RAM. The only true constraint is the disk space for which we suggest at least 200GB of free space to prevent issues.

A.2.4 Software dependencies

CopyKat was developed using Linux Ubuntu 22.04 LTS. We packaged CopyKat with Docker to reduce the burden of satisfying large set of dependencies. Our main software dependencies on the evaluator's machine are the latest `python3`, `docker`, `docker-compose`, `git`, `gcc` and `git-lfs` that can be downloaded on Ubuntu 22.04 LTS. Newer versions of Ubuntu could potentially work, but we did not test them.

A.2.5 Benchmarks

None

A.3 Set-up

As initial step, it is necessary to check if all the software dependencies are installed.

```
sudo apt get install python3 docker.io docker-compose
↪ docker-compose-plugin git git-lfs
```

Then, before cloning our repository, make sure to configure `git-lfs` on the system. Our repository contains few large files that cannot be directly stored on the Github repository.

```
git lfs install
git clone https://github.com/IBM/CopyKat.git
```

Unfortunately `git-lfs` has a monthly limit for download through `git clone`. If such limit is reached, we encourage to download the following files through the web interface.

```
# taint verification
linux/vmlinux

# concolic execution
s2e-data/database/panda.data
s2e-data/kernel/vmlinux
s2e-data/s2e-image/rootfs.ext2
s2e-data/s2e-image/rootfs.ext4.ready
```

A.3.1 Installation

We consider the installation of the two components separately. The two components can be found under the folders `taint_verification` and `concolic_execution_with_corruption` respectively.

Taint verification To successfully run our PANDA analysis, these steps must be followed. First, we need to use *static analysis* to extract some information from the Syzkaller reproducers by running:

```
./01.extract-info.py  --crashes ./repros/ --outfile
↪ db/01.data
```

Then, it is required to compile the reproducers into binary by invoking:

```
./02.compile.py --reports-file db/01.data --path repros/
↪ --outdir ./repros/
```

Now, we need to create the docker container for PANDA and connect to it.

```
cd panda/
docker compose up -d
docker exec -it panda-syz-rrr-1 bash
```

Our PANDA analyses are based on an execution recording of each reproducer. To generate these recordings that will be used to replay later, it is necessary to run:

```
./kdo/run-repros.py --reports-file db/01.data --path
↪ /root/repro/ --outfile db/record.1.json
```

This command will take a considerable amount of time. It needs to boot the QEMU snapshot and then sequentially record executions. The program generates a recording file called `db/record.1.json`. Recordings might fail: grepping the string "err" will reveal recording errors. This does not reflect any fundamental issues with the approach: recording errors occur because our termination detection and timeout logic are not sufficiently sophisticated to capture all cases. We have witnessed that it was sufficient to retry and have created a script to do so. As long as there are still errors in the output file, you can run

```
./kdo/rerun-repros.py --reports-file db/01.data
↪ --record-file db/record.1.json --path /root/repro/
↪ --outfile db/record.2.json
```

To complete the setup, we can extract from the recording information related to the reproducer execution and feed them back into the database.

```
./03.mergeRecordResults.py --infile db/01.data --outfile
↪  db/02.data --recordfile db/record.2.json
```

Concolic execution The setup for concolic execution requires creating the docker container where S2E will be downloaded, patched, compiled and executed.

```
docker build -t s2e-image .
```

This will take approximately 10-15 Minutes. We experienced in our testing a sporadic linking issue for one S2E component when compiled inside `docker`.

```
/usr/bin/ld: cannot find -lglibc-compat-main: No such file
↪  or directory
```

Re-running the exact same command a second time solves the issue. This phase executes cloning inside the docker all the subprojects of S2E, seek them to the right commit hash, patch each with our changes, and compile them. Furthermore, we install our pre-made S2E image that runs our modified kernel. We also create, for each object in the database coming from the taint verification, a new S2E project for which we create a specific configuration file that instruct our S2E plugin on how to approach the analysis. More details can be found in the provided Dockerfile.

A.3.2 Basic Test

Taint Verification We provide here the instructions on how to run a basic test for one reproducer. For instance, for our paper running example 408bf2a23593a83e9ab66cc2982c30d87e52b223 case.

```
./kdo/analysis1/run-analysis.py --record-file
↪  db/record.2.json --repro-id
↪  408bf2a23593a83e9ab66cc2982c30d87e52b223

./04.mergeReplayResults.pass1.py --infile db/02.data
↪  --outfile db/03.data --path panda/out/

./kdo/analysis2/run-analysis.py --record-file db/03.data
↪  --repro-id 408bf2a23593a83e9ab66cc2982c30d87e52b223

./05.mergeReplayResults.pass2.py --infile db/03.data
↪  --outfile db/04.data --path panda/out/

./kdo/analysis3/run-analysis.py --record-file db/04.data
↪  --repro-id 408bf2a23593a83e9ab66cc2982c30d87e52b223

./06.mergeReplayResults.pass3.py --infile db/04.data
↪  --outfile db/05.data --path panda/out/
```

It is also possible to run all the reproducers in bulk by omitting the `-repro-id` argument. That will spin up as many PANDA instances running the analysis in parallel as the machine has cores. The degree of parallelism can be controlled with `-npar`.

Concolic execution To run the concolic execution, we need to first spin and connect to the docker image that we have created during setup

```
docker run -dit --name s2e --restart unless-stopped
↪  s2e-image
docker exec -it s2e bash
```

We provide two scripts to run the tests. The first only runs the analysis on our successfully verified reproducers, as from the tables in the paper (Recommended).

```
python $DATA/scripts/run_successful_ids.py
```

The second one runs all the 222 cases from the PANDA analysis. This considerably increases the time of execution especially on small machines.

```
python $DATA/scripts/run_all.py
```

These scripts are set to run CopyKat on a medium-sized machine (20 cores and 32 GB of RAM). Based on your machine specs, you can edit under `$DATA/scripts/run_*.sh` and set how many parallel jobs–i.e., `Pool(NJOBS)`–your machine can support . We advise to set `N_CORES-1`. However, if you are heavily memory constrained (e.g., <16GB) do not exceed 10 jobs.

It is also possible to run each individual test by hand. For instance, if we want to re-run our running example from the paper:

```
cd 408bf2a23593a83e9ab66cc2982c30d87e52b223
./launch.sh
```

A.4 Evaluation workflow

Taint Verification To evaluate the results of the taint verification phase, the evaluator can inspect the final output file from the latest command. Specifically, `db/05.data` in the example above should include all the information collected on the Panda recording. It is possible to verify such output with the one provided under

```
concolic_execution_with_corruption/s2e-data/database/panda.data
```

which is the complete database for all the objects we have created during our evaluation. `panda.data` is quite large and therefore we suggest the use of file editor that can handle large files (e.g., vim). Alternatively, it is possible to verify the entry related to each reproducer against the one used during Concolic Execution. For instance, to verify the entry of our running example outputted by PANDA, it should match the file:

```
408bf2a23593a83e9ab66cc2982c30d87e52b223/
  408bf2a23593a83e9ab66cc2982c30d87e52b223.output
```

inside the S2E docker container.

Concolic Execution To verify the results of our characterization phase through concolic execution, it is possible to run from another shell connected to the S2E docker container the following commands:

```
find -name report.txt | xargs -I {} grep -l true {}
find -name report.txt | xargs -I {} grep -l true {} | wc
↪ -l
```

These commands can be used to monitor the progress of successfully characterized reproducers. The first returns a list of paths to the successful reports. The second returns just the total number. It is also possible to run the following two commands to generate a sql database and to query it to generate the results we included in our tables in the paper:

```
python $DATA/scripts/process_result.py
bash $DATA/scripts/query_table_final.sh
```

It is possible to directly verify the output of the `query_table_final.sh` command with the tables available in the paper. These commands can be re-run whenever new tests successfully terminate.

Finally, it is possible to inspect individually the various output files of each run. Each project folder will contain a folder `s2e-last` which point to the latest run. This folder contains the following files:

`debug.txt` provides the output of our plugin.

`report.txt` provides general information (e.g., number of constraints, primitive signal etc.) about the test, generated if the analysis does not fail. It can be compared with the one provided in our results on Zenodo, under the archive `constraints.zip`.

`counterfeit-*.txt` provide path-constraints collected at the memory operation or return for the memory area we characterize.

A.4.1 Major Claims

(C1): *Characterization of CCO data-only amplifier objects.*
(C2): *An automated tool, CopyKat, to find and extract characteristics of a CCO.*
(C3): *Identification of 122 CCOs, with 8 exploits demonstrating their practical use.*

A.4.2 Experiments

We designed two experiments to support our claims **C2** and **C3**. We provide below a high level description of the steps for each experiment. For more detailed instructions, please refer to commands listed above, or under the `README`s provided in the Github repository for each stage.

(E1): *[Generate PANDA DB] [10 human-minutes + 24-48 compute-hour + 50GB disk]: This experiment aims to verify the lightweight taint detected during fuzzing and generate a database with detailed information on which elements compose the CCO primitive detected.*

Preparation: Extract the reproducer metadata. Compile the reproducers that should be tested. Create the PANDA image through the docker-compose command. Generate at least one valid execution recording to be replayed in the next phase.
Execution: On the valid recording, invoke the three main PANDA passes in sequence. Integrate after each analysis pass the new output inside the DB through the corresponding merge pass.
Results: The results of this analysis are the latest output files under the folder db. These files would be the input to the concolic execution phase, and can be verified against our `panda.data` db.
To be noticed, our cache name detection is done statically and might not be correct. Whenever we were not able to detect any cache information we fill the entry with `_kmalloc_`. For the cases for which we provide exploits, we manually verified such information and update them accordingly. This can be easily done through `gdb` by calculating the size of the object type under analysis using the provided kernel image. This means that some cache name might differ from the one presented in the paper.
For simplicity and completeness, we rely on our `panda.data` database to run the next step and not the one generated in this test.

(E2): *[Concolic Execution] [1 human-Minute + 1 to 10 compute-hour + 150GB disk]: This experiment runs the validated reproducers under concolic execution and verifies that the primitive is valid after simulated corruption. It provides as output the path-constraints describing the primitive. By default as input it takes the pre-computed PANDA database that we provide. For the sake of reproducing our results, it is not necessary to run/complete the PANDA analysis to test this step.*
Preparation: Create the s2e-image through the `docker build` command.
Execution: Run and connect to the generate s2e-image. Run either manually (`./launch.sh`) or through bulk run (`run_successful_ids.py`) the reproducers. Based on the computation power available, please adjust the Pool number within the script to tailor towards the evaluator's machine. Currently, the script timeouts after 1h. This is to guarantee that the analysis does not get stuck on problematic cases and completes the highest number within a short time. Some cases might require multiple hours or attempts to successfully complete.
Results: The results of this analysis are in the `$hash$/s2e-last` folders. By using the scripts provided, aggregate the results and compare against the paper tables. While the majority ($\sim$106) reproducers should finish within a couple of hours (depending of the computational power). Some take significantly longer and many attempts to successfully be characterize. This

is due to few factors described in the paper like concurrency, inaccurate information gathered at the PANDA stage, or S2E spuriously crashing under certain conditions. These results can be verified against Tables 4, 5 and 6 in the main paper.

A.5 Notes on Reusability

Our artifact is meant to be re-used with minimal changes. The most obvious re-use opportunity is to explore CCOs under another kernel version. This would only require to instrument through our compiler pass a new kernel version, and re-generate the QEMU images and snapshots that are at the base of Syzkaller, PANDA and S2E. This is necessary to guarantee transferrability of information across the analysis (e.g., offsets, functions).

Another quite obvious re-use opportunity of CopyKat concerns the extension to other data-only object patterns. Currently, CopyKat searches and verifies only for CCOs. However, the more data-only objects are discovered, the more patterns can be encoded in out pipeline. This would require changes in how the taint is managed during execution across the fuzzing campaign and the PANDA analysis. Depending on the type of object, changes to our characterization step might be required as well.

You can include in this section any sort of instruction that you believe would help others re-use your artifact, like, for example, scaling down/up certain components of your artifact, working on different kinds of input or data-set, customizing the behavior replacing a specific module/algorithm, etc.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: SoK: On the Fragility of Memory Error Exploit Mitigations

Adriaan Jacobs, Mahmoud Ammar, and Stijn Volckaert

A Artifact Appendix

A.1 Abstract

This artifact contains the Python implementation of the graph-based framework described in Section 3 as well as visualized in Figre 2 in the paper. The artifact implements the memory corruption exploit graph in addition to declarative mappings of defenses to inhibited graph edges. It also provides scripts for automated fragility, interoperability, and portability tests. Additional scripts are also provided for reproducibility, including one that iterates over all mapped defenses and generates a table similar to Table 4 in the paper. The artifact is intended to support the Functional and Results Reproduced badges by allowing reviewers to reproduce the modeled defense mappings and the automated analyses reported in Sections 4, 5, and 6.

A.2 Description & Requirements

The artifact is fully implemented in Python and is straightforward to execute. It requires no compilation, execution of exploits, specific hardware, or custom benchmarks.

The main files of our implementation are:

- `ExploitGraph.py`: The exploit graph model.
- `Explorer.py`: Utility class that tracks inhibited knowledge and edges, and computes reachability queries over an ExploitGraph.
- `Defenses.py`: Contains the list of modeled defenses on the graph, as methods that apply inhibitions to an Explorer instance. It also serves as the command-line fragility/interoperability entry points.
- `generate_defense_table.py`: Iterates through all defenses defined in Defenses.py and evaluates their effectiveness. It outputs Table 4 from the paper.
- `cla_generation.py`: Performs the interoperability evaluation described in Section 6.1 of the paper, between minimal first-level defense sets and Safe Rust.
- `portability_test.py`: Tests the fragility and platform-independence of the modeled defenses by temporarily disabling foundational, assumed protections (like DEP or read-only code) and observing if new attack vectors open up. Described in Section 6.2 of the paper.

A.2.1 Security, privacy, and ethical concerns

The artifact performs abstract graph reachability analyses. No security or privacy risks as it does not generate exploits, run vulnerable programs, disable active security mechanisms, collect data, or access the external network during execution. The scripts write local output files such as `defense.svg`. Reviewers may delete these generated files after evaluation.

A.2.2 How to access

The artifact is archived on Zenodo at https://doi.org/10.5281/zenodo.20315524. The development repository is available at https://github.com/adriaanjacobs/sec26-sok-fragility.

A.2.3 Hardware dependencies

None. The artifact runs on a commodity laptop or desktop with any budget in terms of computing power or storage. The experiments should finish in a few seconds on a typical machine or workstation.

A.2.4 Software dependencies

The artifact was tested on Linux with Python 3.12.3 and the Python `graphviz` package version 0.20.2. The code uses modern Python type syntax, so Python 3.10 or newer is recommended.

A.2.5 Benchmarks

None. The experiments operate only on the graph model and defense mappings included in the artifact.

A.3 Set-up

A.3.1 Installation

Considering the Git repository as the main source of accessing the artifact:

```
git clone https://github.com/adriaanjacobs/
    sec26-sok-fragility.git
cd sec26-sok-fragility
python3 -m venv .venv
. .venv/bin/activate
python3 -m pip install graphviz
```

Listing 1: Artifact Setup Instructions

For the zenodo version, simply download and unzip the zenodo artifact instead, and then start from the `venv` step.

On Debian/Ubuntu systems, install the Graphviz renderer:

```
sudo apt-get update
sudo apt-get install graphviz
```

Listing 2: Desktop Graphic Renderer

A.3.2 Basic Test

Run a single fragility query:

`python3 Defenses.py fragility SoftBound_wo`

The expected results or output should be something similar to the below screenshot.

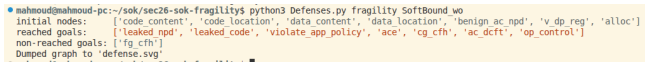


Figure 1: Expected Results when running an automated fragility test for SoftBound Write-only version.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The artifact implements the exploit graph and declarative defense-mapping interface used by the paper to map defenses onto inhibited edges and/or knowledge nodes. This supports the claims and descriptions in Section 3 and 4, and Appendix B. It is reproduced by experiment (E1).

(C2): The artifact automatically evaluates the fragility of mapped defenses and outputs results concerning which assumptions or goals hold and which ones are (partially) violated, as shown in Table 4. This is reproduced by experiment (E2), which mainly depends on experiment (E1).

(C3): The artifact supports and reproduces the interoperability analysis discussed in Section 6.1: Safe Rust fails to compose with several complete first-level defense combinations; It is only interoperable with the combinations with the design-level properties summarized in Table 5 (i.e, these four combinations eliminate all CLA paths when deployed with Safe Rust on the same platform). This is reproduced by experiment (E3).

(C4): The artifact supports and reproduces the portability analysis discussed in Section 6.2: defenses that rely on DEP or read-only code become fragile when such basic defenses are missing in target architectures. This is reproduced by experiment (E4).

A.4.2 Experiments

(E1): Graph modeling, Defense mapping, and a basic fragility test [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment is very similar to the basic test descried in Section A.3.2.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: No further preparation needed.
Execution: Run:
`python3 Defenses.py fragility <defense_name>`

`Defenses.py` includes 50+ mapped defenses. The reviewer can choose to run the automated fragility test of any of these defenses (e.g., `SoftBound`, `LowFat`, `SafeInit`, `DSR`, `DFI`, `SafeStack`, `Clang_IFCC`, etc.) or map new ones and test them.
Results: Figure 2 shows the expected results when running the automated fragility test for `SafeStack`, `Clang_IFCC`, and `CPI` respectively. The printed output shows both reachable and non-reachable attacker goals. Reviewers can also visualize blocked or reachable goals by rendering the resulted `.svg` files using `Graphviz` or any other suitable tool.



Figure 2: Expected Results when running an automated fragility test for SafeStack, Clang-IFCC, and CPI respectively.

(E2): Regeneration of the Defense Table (Table 4) [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment mainly generates a table similar to Table 4 in the paper, covering all mapped defenses. The expected table to be generated by the reviewers should cover defenses shown in Table 4 as well as in Table 6.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: The below commands/definition must be added to the latex environment (e.g., in the preamble file) in order to have the nice visualization inside certain

cells of the generated table.

```
\definecolor{darkgreen}{rgb}{0.0, 0.5, 0.0}
\definecolor{babyblueeyes}{rgb}{0.63, 0.79,
↪ 0.95}
\newcommand{\cmark}{\textcolor{darkgreen}{\larg
↪ e\ding{51}}}
\newcommand{\cmarkbad}{\textcolor{red}{\normals
↪ ize\ding{51}}}
\newcommand{\xmark}{\textcolor{red}{\ding{55}}}
\newcommand{\xmarkgood}{\textcolor{darkgreen}{\
↪ ding{55}}}
\newcommand{\nmarkr}{\textcolor{red}{\ding{108}
↪ }}
\newcommand{\nmarkg}{\textcolor{darkgreen}{\din
↪ g{108}}}
\newcommand{\nmarko}{\textcolor{orange}{\ding{1
↪ 08}}}

\newcommand{\numcircleptrinvalid}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backgray,
    ↪ text=textgray, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backpurple1}{HTML}{76608A}
\definecolor{textpurple1}{HTML}{FFFFFF}

\newcommand{\numcirclewriteabuse}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backpurple1,
    ↪ text=textpurple1, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backblue}{HTML}{DAE8FC}
\definecolor{textblue}{HTML}{000000}

\newcommand{\numcirclereadabuse}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backblue,
    ↪ text=textblue, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backorange}{HTML}{FFE6CC}
\definecolor{textorange}{HTML}{000000}

\newcommand{\numcircleleakage}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backorange,
    ↪ text=textorange, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backblack}{HTML}{000000}
\definecolor{textblack}{HTML}{FFFFFF}

\newcommand{\numcircleexploitdispatch}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backblack,
    ↪ text=textblack, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}

\definecolor{backpink}{HTML}{F8CECC}
\definecolor{textpink}{HTML}{4D4D4D}

\newcommand{\numcircleexploitexecute}[1]{%
  \tikz[baseline=(char.base)]{
    \node[draw=none, fill=backpink,
    ↪ text=textpink, circle, minimum
    ↪ size=13pt, inner sep=0pt] (char)
    ↪ {\footnotesize #1};
  }%
}
```

Execution: Run:

```
python3 generate_defense_table.py > def_tbl.tex
```

Results: A `.tex` file will be generated containing a table similar to Table 4 in the paper covering all mapped defenses.

(E3): Cross-language Interoperability Analysis [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment is very similar to the basic test descried in Section A.3.2.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: No further preparation needed.
Execution: Run:

```
python3 Defenses.py interop --set1 Rust
   --set2 FineIBT intelSHSTK

python3 cla_generation.py > cla_results.txt
```

To run the interoperability test for two selected sets of defenses, the reviewer should execute the first command shown above. In this example, we use the same defense sets as in the paper: Safe Rust as the first set, and FineIBT with Intel Shadow Stack as the second set. To evaluate interoperability between Safe Rust and all modeled first-level defenses, as reported in the paper, and to obtain

results similar to Table 5, the reviewer should execute the second command shown above.
Results: Figure 3 shows the expected results when running the automated interoperability test for `Safe Rust` vs `FineIBT` and `Intel Shadow Stack`, confirming that `Safe Rust` is not interoperable with such Integrity-of-Use defenses. Reviewers can also visualize blocked or reachable goals by rendering the resulted `.svg` files using `Graphviz` or any other suitable tool. We do not show results for the second command line, but expected ones should be in the form of records telling whether `Safe Rust` is interoperable with the selected set of first-level defenses. Out of 55 records, only 4 records should block all CLA paths when deployed with Rust as shown in Table 5 in the paper.



Figure 3: Expected Results when running an automated Interoperability test for Safe Rust vs FineIBT & Intel Shadow stack.

(E4): Portability Analysis [5 human-minutes + less than 1 compute-minute + less than 5 MB disk]:
This experiment is very similar to the basic test descried in Section A.3.2.
How to: Complete the installation steps above and enter the `sec26-sok-fragility` directory.
Preparation: No further preparation needed.
Execution: Run:
`python3 portability_test.py`

Executing the above command runs a reachability analysis query for all mapped defenses under the assumption that these defenses are deployed on platforms lacking W⊕X protection. The obtained results are then compared against the default deployment settings, and the script prints the defenses for which new attack goals become reachable due to the absence of W⊕X.
Results: Figure 4 shows the defenses identified by the portability test. The results indicate that these defenses are not portable to platforms that lack W⊕X protection, since W⊕X is a core assumption in their threat models.

A.5 Notes on Reusability

The artifact is designed to be extended by adding new defense mechanisms to `Defenses.py`. To be mapped, a defense should receive an `ExploitGraph` and an `Explorer`, then marks the graph edges or knowledge nodes it inhibits. Optional annotations such as `@citation`,



Figure 4: Evaluating the Portability of mapped defenses when deploying them in modern architectures, e.g., Nvidia GPUs, that lack W⊕X protection.

`@prettyname`, `@defense_strategy`, `@defense_category`, and `@threatmodel` are consumed by the table-generation and analysis scripts. After adding a defense, reviewers or future users can run `generate_defense_table.py`, `Defenses.py fragility`, and `Defenses.py interop` to evaluate it under different assumptions or deployment scenarios as explained in earlier sections.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: kSFS: Repurposing a Microkernel-like Interface for Fast and Secure In-Kernel Linux File Systems

Dinglan Peng
Purdue University

Pedro Fonseca
Purdue University

A Artifact Appendix

A.1 Abstract

This artifact contains the code and scripts for building kSFS and its dependencies, and for comparing it with other systems.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Running the experiments with this artifact may cause data loss. We suggest using a new machine without important data.

A.2.2 How to access

The artifacts of kSFS can be accessed at https://github.com/rssys/ksfs-artifacts/tree/9aa1c814524ea146df78e2023b9221bd6755574d or https://doi.org/10.5281/zenodo.17973522.

A.2.3 Hardware dependencies

To evaluate this artifact, a server with at least a 24-core CPU, 128 GiB RAM, and an NVMe SSD is required. If the artifact is run on a VM, the `FSGSBASE` CPU feature should be enabled, which can be done by setting CPU model to `host` on QEMU.

A.2.4 Software dependencies

Running the experiments requires a fresh installation of Debian 12 (or newer versions) or Ubuntu 24.04 (or newer versions).

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Download to the artifact and run `install.sh` inside it to install all the dependencies including the kSFS kernel. Alternatively, you can manually build the kernel and install required software following the script. After installation, reboot the server with the installed kernel, and then run `build.sh` to build file systems for kSFS and other comparison systems. Please ignore the errors produced by installing libfuse inside the `fuse-opt` directory. For generating figures, the following Python packages are required: `pandas` and `matplotlib`, which can be installed with `pip`. Edit `experiments/env.sh` to configure the disk and the mount point to use in the experiments. The specified disk will be wiped.

A.3.2 Basic Test

Run the following command to create a file system on a disk and mount it with kSFS: `sudo ./drivers/mount.sh ksfs ntfs <disk> <mount point> --mkfs`. Please specify a disk and a mount point for testing. The disk will be wiped.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): kSFS achieves performance better than or in the same order of magnitude as other systems, including FUSE and Bento, for the work loads described in Section 7 whose results are illustrated in Figure 3, 4, 5, 6 and Table 4, depending on the underlying hardware.

(C2): kSFS can catch the injected faults described in Section 7.5 whose results are illustrated in Table 6.

A.4.2 Experiments

Before running experiments, please remove the existing raw data in `experiments/results` to avoid conflicts.

(E1): Sequential Read/Write Performance [30 human-minutes + 2 compute-hour]: compare the performance of kSFS with other systems on sequential read/write performance for C1.

Execution: Run `run.sh`, `run_bento.sh`, and `run_ext4.sh` in `experiments/fio-sequential`.

Results: Run `fio-sequential.py` in `experiments/figures` to generate Table 4.

(E2): Random Read/Write Performance [30 human-minutes + 10 compute-hour]: compare the performance of kSFS with other systems on random read/write performance for C1.
Execution: Run `run.sh`, `run_bento.sh`, and `run_ext4.sh` in `experiments/fio-rand`.
Results: Run `fio-rand.py` in `experiments/figures` to generate `fig-fio-rand.pdf` for Figure 3.

(E3): Filebench Performance [30 human-minutes + 6 compute-hour]: compare the performance of kSFS with other systems on the filebench benchmark tool for C1.
Execution: Run `run.sh`, `run_bento.sh`, and `run_ext4.sh` in `experiments/filebench`.
Results: Run `filebench.py` in `experiments/figures` to generate `fig-filebench.pdf` for Figure 4.

(E4): Small File Performance [30 human-minutes + 6 compute-hour]: compare the performance of kSFS with other systems on small file operations for C1.
Execution: Run `run.sh`, `run_bento.sh`, and `run_ext4.sh` in `experiments/tar`.
Results: Run `tar.py` in `experiments/figures` to generate `fig-tar.pdf` for Figure 5.

(E5): RocksDB Performance [30 human-minutes + 20 compute-hour]: compare the performance of kSFS with other systems on RocksDB for C1.
Execution: Run `run.sh`, `run_bento.sh`, and `run_ext4.sh` in `experiments/rocksdb`.
Results: Run `rocksdb.py` in `experiments/figures` to generate `fig-rocksdb.pdf` for Figure 6.

(E6): Fault Injection [30 human-minutes + 30 compute-minutes]: test kSFS's fault handling for C2.
Execution: Run `run.sh` in `experiments/fault-injection`.
Results: All outputs in `experiments/results/fault-injection` should contain `fault caught`.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: DMGUARD: Safeguarding Kernels from Physical-Page Use-After-Free Vulnerabilities

Juhee Kim*, Jaeyoung Chung*, Dae R. Jeong†, Byoungyoung Lee†
Seoul National University
{kimjuhi96,jjy600901,dae.r.jeong,byoungyoung}@snu.ac.kr

A Artifact Appendix

A.1 Abstract

This artifact provides the implementation and evaluation environment for DMGUARD, a runtime solution that detects physical-page use-after-free vulnerabilities. The artifact includes the following components:

- Linux kernel v6.6 and Mali GPU Driver r52p0 patches implementing DMGUARD
- Five real CVE PoCs and five synthetic PoCs
- Automated scripts for security and performance evaluation
- A QEMU ARM64 evaluation environment based on Docker

Note that some tests requiring specific hardware or kernel versions are excluded from this artifact. Since the performance evaluation runs on QEMU emulation, absolute numbers may differ from those reported in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact contains PoC code for five CVEs: CVE-2024-1065, CVE-2022-36449, CVE-2025-0072, CVE-2022-33917, and CVE-2024-0671. All of these vulnerabilities have already been publicly disclosed and patched. All code runs only within a QEMU virtual environment, posing no impact on real systems and no security risk to the host system.

A.2.2 How to access

The artifact is available from two sources:

- GitHub: https://github.com/compsec-snu/dmguard
- Zenodo DOI: https://doi.org/10.5281/zenodo.17970502

*These authors contributed equally to this work.
†Corresponding authors

A.2.3 Hardware dependencies

The artifact requires an x86_64 host machine. The minimum requirements are 8GB RAM and 100GB free disk space. We recommend 16GB RAM and 8+ CPU cores for faster kernel builds. No special hardware is required as the artifact uses QEMU-based emulation.

A.2.4 Software dependencies

The artifact requires Linux with Docker Engine and Docker Compose installed. All other dependencies are included in the Docker image and require no additional installation. These include LLVM-19/Clang for kernel builds, QEMU system ARM64, aarch64-linux-gnu cross-compiler, debootstrap, expect, and the source code for Linux kernel and Mali GPU driver.

A.2.5 Benchmarks

The CVE PoC code was collected from public GitHub repositories and Project Zero issue tracker. For performance evaluation, we use LMBench microbenchmark, which is publicly available online. All related code and installation scripts are included in this artifact, allowing evaluators to reproduce results without additional downloads.

A.3 Set-up

A.3.1 Installation

The installation consists of four steps.

Step 1: Register binfmt (run once on host).

```
docker run --rm --privileged \
  multiarch/qemu-user-static --reset -p yes
```

Step 2: Clone repository and build Docker image.

```
git clone \
  https://github.com/compsec-snu/dmguard
cd dmguard
docker compose build    # ~10 minutes
```

Step 3: Start and enter container.

```
docker compose up -d
docker compose exec dmguard-ae bash
```

Step 4: Build kernels and VM image (120 compute-minutes).

```
# Inside container
./ae/scripts/setup-all.sh
```

A.3.2 Basic Test

Run the basic test script to verify that the environment is correctly set up (5 compute-minutes):

```
# Inside container
./ae/scripts/basic-test.sh
```

If VM boot and SCP to VM succeed, a pass message will be displayed. Passing this test indicates that security and performance evaluations can run without issues.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): DMGUARD detects physical-page use-after-free vulnerabilities across all Page and Map Context combinations, including both Free-Before-Unmap and Map-After-Free types. `CONFIG_PAGE_TABLE_CHECK` (i.e., `ptcheck`), in contrast, only detects Free-Before-Unmap with CPU-only contexts and remains vulnerable to race condition bypasses. This is proven by experiment (E1) described in §6.2, with results shown in Table 2.

(C2): DMGUARD exhibits acceptable runtime overhead, though relatively higher than `ptcheck`. This is proven by experiment (E2) described in §6.3, with results shown in Tables 4, 7, and 9.

A.4.2 Experiments

(E1): Security Evaluation (20 compute-minutes): This experiment compares the detection capabilities of `ptcheck` and DMGUARD for GPU driver vulnerabilities and synthetic vulnerabilities reproducible in QEMU.
Preparation: Ensure that the basic test in A.3.2 passes.
Execution: Run the security evaluation script: `./ae/scripts/run-security-eval.sh`
Results: Results can be found in `results/security/table2_security.txt`. Expected results are documented in the repository's `README.md` under (E1). C1 is confirmed if DMGUARD detects all 10 vulnerabilities and `ptcheck` detects only 2 vulnerabilities (CVE-2024-1065 and CVE-2022-36449). Note that this artifact reproduces 10 of the 15 vulnerabilities in the paper's Table 2. The remaining 5 CVEs are excluded because CVE-2023-6241 and CVE-2023-33107 require real Mali/Adreno GPU hardware unavailable in the QEMU emulation environment, and CVE-2024-0582, CVE-2024-47674, and CVE-2024-53096 affect kernel versions other than v6.6.0 used in this artifact.

(E2): Performance Evaluation (120 compute-minutes): This experiment measures the overhead of DMGUARD and `ptcheck` compared to baseline, reproducing Table 4 (LMBench), Table 7 (Boot Time), and Table 9 (Memory Overhead).
Preparation: Ensure that the basic test in A.3.2 passes.
Execution: Run the performance evaluation script: `./ae/scripts/run-performance-eval.sh`
Results: Results can be found in three files: `results/performance/table4_lmbench.txt`, `results/performance/table7_boot_time.txt`, and `results/performance/table9_memory.txt`. Expected results are documented in the repository's `README.md` under (E2). C2 is confirmed if DMGUARD overhead is observed to be approximately 3x that of `ptcheck`. Note that since this artifact runs in a QEMU emulation environment, absolute numbers will be higher than those measured on actual Pixel 8 hardware as reported in the paper. Nevertheless, the overhead remains acceptable and similar trends are maintained. Additionally, due to the instability of the QEMU environment, we use median instead of average, unlike in the paper.

A.5 Notes on Reusability

The DMGUARD kernel patch code is written based on Linux kernel v6.6. Manual modifications are required to apply it to other kernel versions or drivers.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Bridging Bitcoin to Second Layers via BitVM2

A Artifact Appendix

This artifact appendix provides a complete evaluation roadmap for the BitVM2 implementation. It includes BITVM2-CORE (arbitrary computation engine) and BITVM2-BRIDGE (first light-client-based Bitcoin bridge). The artifact demonstrates mainnet validation via transactions executed June 2025, with reproducible performance characteristics on regtest/signet networks.

A.1 Abstract

BitVM2 enables arbitrary computation on Bitcoin through SNARK verification using optimistic computation. This artifact includes: (1) a Groth16 SNARK verifier compiled to Bitcoin Script (∼900MB, chunked to <4MB pieces); (2) complete peg-in/peg-out bridge protocol; (3) 1-of-n trust minimization through covenant emulation; (4) mainnet validation via June 2025 transactions; and (5) performance characteristics with <8 hour dispute resolution and <0.15 BTC fees. All major claims are reproducible on Bitcoin regtest/signet.

A.2 Description & Requirements

This section describes the environment required to evaluate the BitVM2 artifact.

A.2.1 Security, privacy, and ethical concerns

Bitcoin regtest mode: The artifact runs Bitcoin Core in regtest (regression test) mode, creating a private local blockchain. No real funds are used during evaluation.

Resource usage: Building the Groth16 verifier requires substantial memory (16GB+ RAM). Some tests generate large files (up to 4MB Bitcoin Script files).

Network activity: Requires internet connectivity to download dependencies (Rust crates, Risc0 toolchain). Bitcoin Core will bind to network ports for the regtest node.

A.2.2 How to access

Repository: https://github.com/bitvm/bitvm

Permanent archive: https://doi.org/10.5281/zenodo.17949747

The artifact is permanently archived on Zenodo with DOI 10.5281/zenodo.17949747, ensuring long-term availability as required for the Artifacts Available badge.

A.2.3 Hardware dependencies

Minimum requirements:

- **CPU**: x86_64 or ARM64 (Apple Silicon supported)
- **RAM**: 16GB minimum, 24GB recommended
- **Disk**: 10GB free space

Tested configurations: Ubuntu 22.04 (x86_64), macOS (ARM64)

A.2.4 Software dependencies

Required: Rust 1.70+ (per `rust-toolchain.toml`), Cargo, Git, Bitcoin Core 25.0+, Risc0 toolchain.

Optional: Docker (containerized regtest), CUDA toolkit (NVIDIA GPUs).

All dependencies are available via standard package managers. Detailed installation in Section 1.3.1.

A.2.5 Benchmarks

Bitcoin blockchain data generated in regtest mode. No external datasets required. Mainnet validation uses publicly accessible Bitcoin mainnet transactions from June 2025.

A.3 Set-up

A.3.1 Installation

Step 1: Clone repository

```
git clone https://github.com/bitvm/bitvm
cd bitvm
```

Step 2: Build (requires 16GB+ RAM)

```
cargo build --release
```

Step 3: Install Bitcoin Core

```
cd regtest && ./install.sh && cd ..
```

Step 4: Configure (edit with RPC credentials)

```
cp .env.sample .env
```

Step 5: Install Risc0 toolchain

```
cargo install cargo-risczero
cargo risczero install
```

A.3.2 Basic Test

Test 1: Verify build

```
cargo test --lib
```

Expected: Unit tests pass (some may be ignored by default for performance reasons).

Test 2: Start Bitcoin regtest

```
cd regtest && ./start.sh
bitcoin-cli -regtest getblockchaininfo
```

Expected: Shows `"chain": "regtest"` and `"blocks": 0`.

Test 3: Run basic script tests

```
cargo test --release u32
```

Expected: Tests for u32 Bitcoin Script primitives pass.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BITVM2-CORE enables arbitrary computation on Bitcoin by compiling a Groth16 SNARK verifier to Bitcoin Script (∼900MB), chunked into 379 pieces (90% <3MB). Proven by (E1); described in Abstract & Section 3; results in Section 7.

(C2): Complete peg-in/peg-out bridge protocol with dispute resolution. Proven by (E2), (E3), (E4); described in Section 5; transaction graphs in Figure 1.

(C3): Trust minimized to 1-of-n (existential honesty) via covenant emulation. Proven by (E5); described in Section 2 & Figure 1.

(C4): Production-level implementation validated on Bitcoin mainnet (June 3, 2025). Proven by (E6); described in Abstract & Section 7.

(C5): Performance: <8h dispute, <0.15 BTC fees, 90% scripts <3MB. Proven by (E1), (E6); results in Section 7, Figures 8-9.

A.4.2 Experiments

(E1): BITVM2-CORE Functionality [5 min + 1GB disk]
Verify Groth16 SNARK verifier compiles to Bitcoin Script and is chunked appropriately.
Preparation: `cd bitvm`
Execution: `cargo test --release groth16 && cargo test --release chunk`
Results: Tests pass. Groth16 ∼900MB Script. 379 chunks, 90% <3MB, range 500KB-3.99MB.
Supports: (C1), (C5)

(E2): Peg-in Protocol [5 min]
Verify successful peg-in (deposit) to bridge.
Preparation: Ensure regtest running: `cd regtest && ./start.sh`
Execution: Follow `DEMO_INSTRUCTIONS.md` Section: "Peg-in Demo"
1. Generate deposit address
2. Fund deposit transaction
3. Confirm deposit on regtest
4. Verify bridge accepts deposit

Results: Deposit confirms in regtest. Bridge records successfully. Verify via Esplora (localhost:5000).
Supports: (C2)

(E3): Peg-out Happy Path [5 min]
Verify successful peg-out (withdrawal) with honest operator.
Preparation: Complete peg-in from (E2).
Execution: Follow `DEMO_INSTRUCTIONS.md` Section: "Peg-out Happy Path"
1. User initiates peg-out request
2. Operator processes withdrawal
3. Confirm peg-out transactions on regtest

Results: All transaction types confirm: Kick-off, Take, Peg-out. User receives funds. Tx sizes <17 vkB.
Supports: (C2), (C5)

(E4): Dispute Resolution [10 min]
Verify dispute mechanism penalizes dishonest operator.
Preparation: Complete peg-in from (E2).
Execution: Follow `DEMO_INSTRUCTIONS.md` Section: "Peg-out Unhappy Path"
1. Operator submits incorrect claim
2. Verifier challenges the claim
3. Operator posts Assert transaction
4. Verifier posts Disprove transaction

Results: Challenge mechanism works. Dispute resolves within timeout. Dishonest operator penalized. Assert & Disprove txs confirm.
Supports: (C2), (C5)

(E5): Trust Model Verification [10 min]
Verify 1-of-n trust minimization through code review.
Preparation: None (code review only).
Execution:
1. Review `bridge/src/contexts/` directory
2. Examine presigned tx structure in `bridge/src/transactions/`
3. Verify signer role (active only during setup)
4. Verify operator role (single honest party ensures liveness)
5. Verify challenger role (permissionless)

Results: Covenant emulation via presigned txs confirmed. Signers only active during setup. Single honest signer prevents theft. Single honest operator ensures liveness. Anyone can challenge.
Supports: (C3)

(E6): Mainnet Validation [5 min]
Verify actual Bitcoin mainnet transactions from June 2025.
Preparation: None (verification only).
Execution: Verify transactions via `mempool.space`:
1. Claim: `6b096f35...d4d147a`
2. Challenge: `49f3425b...5d80581`
3. Assert: `32129c9b...08711f`
4. Disprove: `8ecfefd4...510249`

Results: All txs confirmed on mainnet. Settlement: 42 blocks (7h 36min). Total cost: 14.9M sat (∼$16k). Largest tx: ∼4MB. Matches claims: <8h, <0.15 BTC.
Supports: (C4), (C5)

A.5 Notes on Reusability

Extending: Custom circuits (modify `bitvm/src/groth16/`), different networks (edit `.env`), reuse Script primitives (`bitvm/src/`), adjust bridge parameters (`bridge/src/`).

Limitations: On-chain light client (Section 4) NOT implemented. Large txs (>400kB) require custom relay infrastructure.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Can we estimate privacy vulnerability of individual records? Towards Mitigating Attribute Inference Attacks on ML Models

Ehsanul Kabir
Pennsylvania State University

Najrin Sultana
Pennsylvania State University

Ninghui Li
Purdue University

Shagufta Mehnaz
Pennsylvania State University

A Artifact Appendix

A.1 Abstract

Attribute inference attacks pose a serious privacy risk to machine learning (ML) models by enabling adversaries to infer sensitive attributes of individual records from model behavior. This paper investigates whether privacy vulnerability from attribute inference can be estimated at the level of individual records and used to better understand and mitigate attribute inference attacks. From an adversary's perspective, we introduce NeighVE, a vulnerability estimation method that quantifies record-level susceptibility by computing neighborhood angular difference over synthetically generated neighbors. From the defender's perspective, we propose AttriVET, a vulnerability estimation technique that leverages neighborhood similarity to identify records that are vulnerable to attribute inference. Building on these insights, we introduce VESL, a defense mechanism that uses vulnerability-aware subspace learning to mitigate attribute inference attacks while preserving model utility. For artifact evaluation, our goals are to ensure reproducibility of the experiments, validate the implementation of the proposed vulnerability estimation methods and defenses, and enable further exploration of attribute inference risks in ML systems. The provided artifacts include datasets, a modular codebase, and detailed Jupyter notebooks that reproduce the paper's key findings on the neighborhood-driven nature of record-level privacy vulnerability, the use of vulnerability estimation from adversarial and defensive perspectives, and the effectiveness of VESL in mitigating attribute inference attacks. These artifacts aim to support open science practices, ensuring accessibility, transparency, and reusability for the research community.

A.2 Description & Requirements

This section outlines the experimental setup, including security considerations, access instructions, software dependencies, hardware requirements, and benchmark datasets for reproducibility.

A.2.1 Security, privacy, and ethical concerns

The datasets used in the attacks are anonymized and publicly available; therefore, there is no direct security or privacy risk arising from the release or execution of this artifact. As discussed in the paper's ethical considerations, the techniques implemented in this artifact—particularly vulnerability estimation and attribute inference attacks—could be misused if applied irresponsibly. However, the artifact is released strictly for research, auditing, and defensive purposes, and does not introduce new attack capabilities beyond those already established in prior work. The artifact adheres to the FAIR principles: it is findable via a public repository, accessible without restrictive licensing, interoperable through standard data formats and widely used ML libraries, and reusable through comprehensive documentation and clear modular design. We encourage evaluators and future users to follow responsible research practices and applicable ethical guidelines when using this artifact in privacy-sensitive contexts.

A.2.2 How to access

The artifact is available at `https://doi.org/10.5281/zenodo.17905132`. Please refer to 'Version v2' of the artifact for the most up-to-date version.

A.2.3 Hardware dependencies

None

A.2.4 Software dependencies

The artifact requires a Linux, macOS, or Windows operating system with Python 3.9.16 or later. A virtual environment can be set up using Conda or Python's `venv`. All necessary dependencies are listed in `requirements.txt` and can be installed using `pip install -r requirements.txt`. Running Jupyter notebooks requires installing `notebook` via `pip`

`install notebook`. No proprietary software is required, and all dependencies are open-source and can be installed using standard package managers.

A.2.5 Benchmarks

The experiments in this artifact rely on the following datasets: `census19.csv`, which contains the Census-19 dataset, and `Adult_35222.csv` and `Adult_10000.csv`, which contain the Adult dataset partitions for training and testing, respectively. Additionally, the Texas-100X dataset is required. To ensure reproducibility, we have attached a preprocessed version of the dataset with the filename `texas_100_cleaned.csv`.

A.3 Set-up

A.3.1 Installation

To install and set up the artifact, first create a virtual environment using Conda or Python's `venv` and activate it. Then, install all required dependencies using `pip install -r requirements.txt`. The specific versions of the listed packages that were used in implementation are included in `requirements_w_version.txt` for reproducibility. To install those exact versions, run: `pip install -r requirements_w_version.txt`.
To run Jupyter notebooks, install `notebook` via `pip install notebook`.

A.3.2 Basic Test

To verify the setup, open and run all cells in the Jupyter notebook `basic_test.ipynb`. This notebook loads and preprocesses the Adult dataset for different experiment scenarios. A successful run will produce the first five rows of the training data.

A.4 Evaluation workflow

This section outlines the steps required to evaluate the artifact, validate its functionality, and reproduce the key results presented in the paper. It consists of two parts: Major Claims, which enumerate the core findings supported by the experiments, and Experiments, which describe the operational steps needed to test these claims. Each experiment is linked to a corresponding claim and includes details on execution, estimated compute time, and expected outcomes.

A.4.1 Major Claims

(C1): AttriVET can accurately identify vulnerable records across all three datasets under CSMIA and LOMIA. This is proven by the experiment (E1) described in section 6.2 of the main paper whose results are presented in Table 2.

(C2): VESL can mitigate attribute inference attacks across all three datasets and reduce it below that of imputation attack performance while incurring a small loss in utility and preserving fairness. This is proven by the experiment (E2) described in section 6.3 of the main paper whose results are presented in Tables 3 to 6. Tables 3 and 4 reports attack performance with defense under CSMIA and LOMIA respectively. Table 5 reports the accuracy of the model on test set and Table 6 reports the fairness metrics Equalized Odds Difference (EOD) and Demographic Parity Difference (DPD) across various senstive attribute values.

(C3): NeighVE can identify subset of training records with much higher attack performance than global average for Adult dataset but not for Census-19 dataset. Furthermore, when NeighVE is applied on a VESL-trained model, it becomes ineffective in identifying a more vulnerable attack subset. This is proven by the experiment (E3) described in section 4.1 and section 6.4 of the paper whose results are presented in Figures 1 to 3.

A.4.2 Experiments

(E1): [1 compute-hour]: This experiment evaluates the performance of the proposed individual record-level vulnerability estimation tool, AttriVET, on the Census19, Adult, and Texas-100X datasets under attribute inference attacks. The results are expected to demonstrate high accuracy, precision, recall, and F1-score in identifying vulnerable records.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `attriVET_evaluation.ipynb` or run the python script from terminal: `python attriVET.py`
Results: The last two cells of the notebook and the python script are expected to generate results similar to Table 2.

(E2): [5 compute-hour]: This experiment evaluates the performance of the proposed defense against attribute inference attacks, VESL, on the Census19, Adult, and Texas-100X datasets. Performance is measured across three categories—attack performance reduction, model accuracy on the test set, and fairness across sensitive attribute values. In terms of attack performance reduction, the results are expected to show that both CSMIA and LOMIA achieve lower performance than the imputation attack in the presence of both VESL variants. The results are also expected to show that VESL-trained models achieve test accuracy comparable to models trained without any defense. Finally, the results are expected to show no degradation in the fairness metrics EOD and DPD.
Preparation: No additional preparation required.
Execution: Run all cells in the Jupyter notebook `vesl_evaluation.ipynb` or run the python script from

terminal: `python vesl.py`

Results: The python script is expected to generate results similar to Tables 3 to 6. The notebook cell 8 is expected to generate attack performance results similar to Tables 3 and 4. Cell 9 is expected to generate model test accuracy results similar to Table 5. Cell 10 is expected to generate fairness evaluation results similar to Table 6.

(E3): [4 compute-hour]: This experiment evaluates the performance of the adversar-side vulnerability estimation tool, NeighVE, on the Census19 and Adult datasets on models trained with no defense and models trained with VESL. The performance is reported as the proportion of correctly inferred records in the subset of records with top 25% n.a.d. values. The results are expected to show a high proportion of correctly inferred records for Adult dataset but not for Census-19 dataset. The results are also expected to show that when n.a.d. values are computed on a VESL-trained model, the proportion of correctly inferred records for Adult dataset is reduced to the global average.

Preparation: No additional preparation required.

Execution: Run all cells in the Jupyter notebook `neighve_evaluation.ipynb` or run the python script from terminal: `python neighve.py`

Results: The python script is supposed to generate plots similar to Figures 1 to 3 and save them to the folder `Figures`. Notebook cells 8 and 9 are expected to generate n.a.d. histogram plots similar to Figure 1. Cell 10 is expected to generate barplots similar to Figure 2. Cells 11 and 12 are expected to generate n.a.d. histogram plots using CSMIA and LOMIA respectively with no defense models on Census-19 dataset similar to Figures 3(a) and 3(b). Cells 13 and 14 are expected to generate n.a.d. histogram plots using CSMIA and LOMIA with VESL-trained models on Adult dataset similar to Figures 3(c) and 3(d).

A.5 Notes on Reusability

In future work, our implemented codebase can serve as a foundation for exploring additional aspects of attribute inference attacks and for developing and evaluating new defense strategies. The artifact README includes a detailed mapping of the codebase, outlining the role and functionality of each file and directory.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Five Queries Are Enough: Query-Efficient and Surrogate-Free Membership Inference Attacks on RAG via Entailment

Nguyen Linh Bao Nguyen[1], Wanlun Ma[1], Viet Vo[1]*, Alsharif Abuadbba[2], Minghong Fang[3], Jun Zhang[1], and Yang Xiang[1]
[1]*Swinburne University of Technology, Australia*
[2]*CSIRO, Australia*
[3]*University of Louisville, USA*

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation and evaluation scripts for MEntA, a membership inference attack against Retrieval-Augmented Generation (RAG) systems. The artifact includes MEntA, four baseline attacks (IA-MIA, DCMI, MBA, and S2-MIA), input/output modification defenses, query-detection defenses, preprocessed BEIR datasets, and Apptainer scripts for reproducing the main experiments.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact evaluates membership inference attacks on public BEIR datasets only: NFCorpus, SCIDOCS, and TREC-COVID. It does not attack any deployed service or private database. The online phase uses the OpenAI API for query generation, summarization, query paraphrasing, and GPT-based detection; evaluators should use their own API key and monitor API cost. The offline phase runs local HuggingFace models and may require GPU resources. No destructive system-level actions are performed.

A.2.2 How to access

The artifact is archived at `https://doi.org/10.5281/zenodo.20335970`. Download and unpack the archive, then follow the setup steps below from the artifact root directory.

A.2.3 Hardware dependencies

VRAM requirements vary by experiment and were measured on NVIDIA A100 GPUs:

- Smoke test: approximately 40 GB VRAM;
- Full MEntA evaluation: approximately 75 GB VRAM;
- Baseline evaluations (`IA-MIA`, `DCMI`, `MBA`, and `S2-MIA`): typically more than 100 GB VRAM;

A smaller `RAG_GENERATOR_MODEL` may be used for experiments if `microsoft/phi-4` does not fit, though results may differ from those reported in the paper.

A.2.4 Software dependencies

The main software dependency is Apptainer. Install Apptainer by following the official installation guide: `https://apptainer.org/docs/admin/main/installation.html`. We used Apptainer version 1.3.6 for our implementation.

The artifact provides an `Apptainer.def` file. The container installs the Python dependencies from `requirements.txt`. Evaluators also need an OpenAI API key for online stages, and a HuggingFace token for gated models, if such models are selected.

A.2.5 Benchmarks

The archive ships with `./data/` already populated. It contains preprocessed BEIR corpora for NFCorpus, SCIDOCS, and TREC-COVID, including member/non-member splits, FAISS indices, benign queries, perturbed corpora for DCMI, and per-document summaries used by MEntA retrieval boosting. To refresh or extend datasets, for example to add another retriever index, follow the instruction of setting up data in A.3.1.

A.3 Set-up

A.3.1 Installation

If the cluster requires Apptainer cache or temporary directories to be placed on project storage, copy and edit the provided setup script before building the image:

*Corresponding author: `vvo@swin.edu.au`

```
cp example_apptainer_setup.sh apptainer_setup.sh
# edit APPTAINER_CACHEDIR and APPTAINER_TMPDIR
source apptainer_setup.sh
```

From the artifact root, build the Apptainer image and create the local configuration file. Please note that `apptainer build` typically requires root or fakeroot on many clusters:

```
apptainer build menta.sif Apptainer.def
cp .env.example .env
```

Edit `.env` before running any experiment. For the reported full experiments using `CohereLabs/c4ai-command-r7b-12-2024`, we recommend evaluators to keep the configuration in this file. At minimum, evaluators should set the OpenAI API credentials, HuggingFace token, and HuggingFace cache.

```
OPENAI_API_KEY=...
HF_TOKEN=...
HF_HOME=/absolute/path/to/hf_cache
```

Define the Apptainer helper used by the scripts:

```
apptainer_run() {
  local hf_home
  hf_home="$(set -a; source .env; printf '%s' "$HF_HOME")"

  apptainer exec --nv \
    --env-file .env \
    --bind "$PWD:/workspace" \
    --bind "$hf_home:$hf_home" \
    menta.sif "$@"
}
export -f apptainer_run
```

Download the configured HuggingFace models:

```
apptainer_run bash scripts/setup_model.sh
```

The archive already includes `./data/` and rebuilding data is optional. If needed, run the online data preparation on a networked node first and wait for it to finish successfully. Only then run the offline indexing phase inside a GPU allocation:

```
apptainer_run bash scripts/setup_data.sh --online
# run this only after the online phase has finished
apptainer_run bash scripts/setup_data.sh --offline
```

A.3.2 Basic Test

The smoke test checks the full workflow on a small copied dataset. Run the online phase first and wait for it to finish successfully before starting the offline phase, because the offline phase consumes the queries, paraphrases, summaries, and detector inputs produced by the online phase.

```
apptainer_run bash scripts/run_test_mode.sh --online
# run this only after the online smoke-test phase has finished
apptainer_run bash scripts/run_test_mode.sh --offline
```

The online command executes OpenAI/API stages, while the offline command builds the small FAISS index and runs local GPU stages. A successful test ends with:

```
Smoke-test online phase finished.
Smoke-test offline phase finished.
```

Smoke-test outputs are written under `results/test/`.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1) Under a five-query budget, MEntA achieves the strongest overall membership inference AUC among the evaluated attacks (IA, MBA, S^2MIA, and DCMI). This is the main effectiveness result reported in Table 3.

(C2) MEntA remains effective under input and output modification defenses, including Differential Privacy (DP)-style perturbation, context re-ranking, prompt instruction, and query paraphrasing. These defended-setting results are reported in Table 6.

(C3) MEntA remains robust against input detection defenses. The GPT-4-based detector and Mirabel similarity detector are evaluated in Tables 7 and 8. Although Mirabel can detect a portion of attack queries, it also produces a high false-positive rate on benign queries, making it impractical as a reliable deployment-time defense.

A.4.2 Experiments

The reproduced numerical results may differ slightly from those reported in the paper due to randomness, model/runtime differences, and API nondeterminism, but the overall claims and conclusions should remain consistent.

The main evaluation workflow is `scripts/run_all.sh`. After `.env` is configured, this script runs the attack comparison, defended attack settings, and input detection defenses in one online/offline workflow.

For the full reproduction, configure `.env` with the three datasets, the Cohere generator model, the five-query budget, and all attack/defense modes:

```
DATASETS="BeIR_nfcorpus BeIR_scidocs BeIR_trec-covid"
RAG_GENERATOR_MODEL=CohereLabs/c4ai-command-r7b-12-2024
NUM_QUERIES=5
MENTA_MODES="default dp rerank prompt_instruction paraphrase"
IA_MODES="default dp rerank prompt_instruction paraphrase"
DCMI_MODES="default dp rerank prompt_instruction paraphrase"
MBA_MODES="default dp rerank prompt_instruction paraphrase"
S2MIA_MODES="default dp rerank prompt_instruction paraphrase"
FORCE=0
```

Then run the online phase on a networked node and the offline phase inside a GPU allocation. The online phase must complete successfully before the offline phase is started, because the offline phase consumes the queries, paraphrases, summaries, and detector inputs produced by the online phase:

```
apptainer_run bash scripts/run_all.sh --online
# run this only after the online phase has finished
apptainer_run bash scripts/run_all.sh --offline
```

(E1) Five-query attack comparison. This experiment validates C1. In the workflow above, `run_all.sh` first runs the default mode for MEntA and all baselines. The online phase generates attack queries and other API-dependent artifacts. The offline phase performs retrieval, local RAG generation, scoring, and evaluation. MEntA results are written under `results/MEntA/<dataset>/evaluation/`. Baseline results are under `results/IA-MIA/`, `results/DCMI/`, `results/MBA/`, and `results/S2-MIA/`. The AUC values in these default evaluation files correspond to Table 3.

(E2) Input and output modification defenses. This experiment validates C2. Because the mode variables above include `dp`, `rerank`, `prompt_instruction`, and `paraphrase`, `run_all.sh` runs each defense as a separate pass for every attack. Modes are not combined within one attack run. The resulting defense-specific outputs are stored in directories such as `evaluation_dp/`, `evaluation_rerank/`, `evaluation_prompt_instruction/`, and `evaluation_paraphrased/`. These files provide the values summarized in Table 6.

(E3) Input detection defenses. This experiment validates C3. After the attack query files have been generated, `run_all.sh` invokes `scripts/run_input_detection_defenses.sh`. During the online phase, it builds the unified `queries_dataset.jsonl` files and runs the GPT-based detector. During the offline phase, it runs the Mirabel-style similarity detector. GPT detection outputs are written under `results/gpt_detection/`; Mirabel outputs are written under `results/mirabel_detection/`. The resulting recall and false-positive-rate summaries correspond to Tables 7 and 8.

A.5 Notes on Reusability

The main configuration interface is `.env`. Evaluators can change datasets, model names, retrievers, query budgets, top-k, and modes without editing Python code. The scripts support resume-by-default behavior: completed stage outputs are skipped unless `FORCE=1`. OpenAI calls can use either the Batch API or sequential single-request inference by setting:

```
OPENAI_INFERENCE_MODE=batch
OPENAI_INFERENCE_MODE=single
```

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Membership Inference Attacks on Tokenizers of Large Language Models

Meng Tong[1]* Yuntao Du[2]* Kejiang Chen[1]✉ Weiming Zhang[1] Ninghui Li[2]

[1]*University of Science and Technology of China* [2]*Purdue University*
*Equal Contribution ✉Corresponding Author

A Artifact Appendix

A.1 Abstract

Membership inference attacks (MIAs) are widely used to assess the privacy risks associated with machine learning models. However, when these attacks are applied to pre-trained large language models (LLMs), they encounter significant challenges, including mislabeled samples, distribution shifts, and discrepancies in model size between experimental and real-world settings. To address these limitations, we introduce tokenizers as a new attack vector for membership inference. Specifically, a tokenizer converts raw text into tokens for LLMs. Unlike full models, tokenizers can be efficiently trained from scratch, thereby avoiding the aforementioned challenges. In addition, the tokenizer's training data is typically representative of the data used to pre-train LLMs. Despite these advantages, the potential of tokenizers as an attack vector remains unexplored. To this end, we present the first study on membership leakage through tokenizers and explore five attack methods to infer dataset membership. We conduct extensive experiments verify the dataset-level membership leakage from tokenizers.

A.2 Description & Requirements

A.2.1 Hardware Requirements

The experiments were conducted on a system equipped with at least **32GB of RAM** to support the training of shadow tokenizers and dataset sampling.

A.2.2 Software Requirements

The artifact is designed to run on Linux-based distributions. The following software environment is required:

- **Package Manager:** Conda (Miniconda or Anaconda)
- **Language:** Python 3.12
- **Libraries:** HuggingFace `tokenizers`, `transformers`, and the dependencies specified in `requirements.txt`.

Environment Setup To replicate the environment, execute the following commands:

```
conda create -n MIA python=3.12
conda activate MIA
pip install -r requirements.txt
```

A.2.3 Security, privacy, and ethical concerns

The introduction of a new attack vector for MIAs could be used to infer personal information and lead to privacy violations. However, our experiments solely utilize public,non-sensitive datasets collected by Google. We emphasize that our proposed attacks are never intended to facilitate or encourage the privacy leakage from LLMs.

A.2.4 How to access

We are committed to upholding the principles of open science by making our findings freely accessible. Our codes are available at https://zenodo.org/records/18080228. It is recommended to download the files on linux computer.

A.2.5 Hardware dependencies

None.

A.2.6 Software dependencies

The experimental environment is managed using Conda. Users are required to have Conda (or Miniconda) installed. The installation of conda are referred to https://www.anaconda.com/download. The specific Python dependencies are listed in `requirements.txt`, which is available at https://zenodo.org/records/18080228.

A.2.7 Benchmarks

We utilize real-world web data from the C4 corpus in our evaluation. We have provided the involved datasets at https://zenodo.org/records/18080228.

A.3 Set-up

A.3.1 Installation

To replicate the environment, execute the following commands to set up the Python environment and install the required dependencies:

```
conda create -n MIA python=3.12
conda activate MIA
pip install -r requirements.txt
```

A.3.2 Basic Test

After the environment is correctly configured, run the following command in the terminal to implement a membership inference attack via vocabulary overlap:

```
python train_target_tokenizer.py
python train_shadow_tokenizer.py
python mia_via_vocabulary_overlap.py
```

The vocabularies of target tokenizers are stored in the `trained_tokenizer` directory. Also, the generated shadow tokenizers are saved in the `shadow_tokenizer` directory. The evaluation results and performance of membership inference are saved in the `infer_results` directory.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Tokenizers are vulnerable to advanced membership inference attacks, as demonstrated by experiment E1 (Section 5.2) and the results in Table 3. For instance, MIA via Vocabulary Overlap and MIA via Frequency Estimation yield AUC scores exceeding 0.70 when evaluated on a target tokenizer with a vocabulary size of 200,000.*

(C2): *Expanding a tokenizer's vocabulary increases its susceptibility to membership inference attacks. This correlation is validated by experiment (E1) and the corresponding results reported in Table 3.*

A.4.2 Experiments

(E1): *[Comparison of MIAs against target tokenizers] [30 human-minutes + 24 compute-hour + 30GB disk]:*
Preparation: To set up the experimental environment, create a dedicated Conda environment and install the dependencies as follows:

```
conda create -n MIA python=3.12
conda activate MIA
pip install -r requirements.txt
```

The datasets are in the `website_data/` folder.

Execution: The experiment follows training target tokenizers, training shadow tokenizers, and executing the membership inference attacks:

```
python train_target_tokenizer.py
python train_shadow_tokenizer.py
python mia_via_compression_rate.py
python mia_via_vocabulary_overlap.py
python mia_via_frequency_estimation.py
python mia_via_merge_similarity.py
python mia_via_naive_bayes.py
```

The training scripts will generate tokenizers in the `trained_tokenizer` and `shadow_tokenizer` folders.
Results: Upon completion of the attack scripts, all experimental outcomes, including inference accuracy and performance metrics, are automatically aggregated. The results are saved in the `infer_results` directory.

A.5 Notes on Reusability

The membership inference attacks can also be executed on custom datasets. To perform this, users should place the target dataset in the `website_data` directory.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Distributed Synthesis of Differentially Private Tabular Datasets

Yucheng Fu*, Tianyao Gu†, Elaine Shi†, Tianhao Wang*
*University of Virginia, †Carnegie Mellon University
{zdp8uu, tianhao}@virginia.edu, tianyaog@andrew.cmu.edu, elaineshi@cmu.edu

A Artifact Appendix

A.1 Abstract

This artifact is the implementation of the paper *Distributed Synthesis of Differentially Private Tabular Datasets*. Our goal with the artifacts is to provide the source code and scripts to realize the claimed functionality described in the paper, and reproduce the experimental results.

In our paper, we proposed two primitives to facilitate the differentially private tabular synthesis in the distributed setup: a two-way marginal computation protocol and a noise generation protocol. Correspondingly, our artifact contains a separate implementation of these two primitives that might be of independent interest in other applications. We also provide the implementation of our end-to-end synthesis protocol that utilizes these primitives.

We provide all our experiment code either as one-line commands or as a simple script file to make the results simpler to reproduce. Our main goal with the experiments is not to reproduce the exact same numbers as we have in our figures, as it will vary greatly with the hardware used, but to show that the speedup we obtain, as well as the asymptotic behavior, is similar to the shown in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. Our code does not have any destructive steps, nor does it disable any security mechanisms.

A.2.2 How to access

Our artifact is available in Zenodo: https://doi.org/10.5281/zenodo.18218427

A.2.3 Hardware dependencies

In terms of infrastructure, all of our experiments were run on the same AWS instance type c5a.8xlarge, with 32 CPU of x86_64 architecture, 64 GiB RAM, and 64 GiB storage. The OS is Ubuntu Server 22.04 LTS (HVM), SSD Volume Type. We suggest using a similar OS and at least 32 GB free disk space. Otherwise, the automatic environment setup script might fail.

A.2.4 Software dependencies

Our artifact does not depend on any pre-installed software. All dependencies (Conda, Docker, Python environments, etc.) are automatically installed by the setup script, and most of the experiments are meant to be run under Docker.

A.2.5 Benchmarks

We use the UCI Adult dataset https://archive.ics.uci.edu/dataset/2/adult, which is licensed under a Creative Commons Attribution 4.0 International (CC BY 4.0) license: https://creativecommons.org/licenses/by/4.0/legalcode.

A.3 Set-up

A.3.1 Installation

Use terminal to navigate into the root path of the artifact. We provide a single-line command to set up the environment.

```
./run_exp.sh setup
```

Running the script "./run_exp.sh" given the "setup" parameter will install all the dependencies of our experiment, including Conda, Docker, and the necessary Python environment, etc. This script does not assume any pre-installed package and can make sure the later experiments can be run correctly even in a newly initialized instance.

A.3.2 Basic Test

After setup, start the Docker containers and navigate to the artifact root directory:

```
cd docker && sudo docker-compose up -d && cd ..
```

We provide two lightweight test scripts to verify the installation without running the full experiments.

Test 1: Marginal computation. (test_marginal.sh) This script runs the two-way marginal computation on the default demo dataset (`adult_small.csv`) through the full DPF protocol: key generation on the client, key evaluation on 3 parties, share collection, and reconstruction:

```
./test_marginal.sh
```

Expected output upon success:

```
==========================================
TEST PASSED
  - Marginal shares collected from 3 parties
  - Reconstruction successful
  - Total sum = 100 (matches 100 records)
==========================================
```

Test 2: Secure noise generation (test_noise.sh). This script runs a single secure noise generation protocol across 3 parties using MP-SPDZ. It accepts three optional arguments: noise type, number of samples, and scale parameter.

```
./test_noise.sh [dgauss|dlap] [n_samples]
                [scale]
```

The defaults are `dgauss`, `4096`, and `10`, respectively. To quickly test with the default parameters:

```
./test_noise.sh
```

Expected output upon success:

```
==========================================
TEST PASSED
  - Noise generation completed successfully
  - Time: <time>s
  - Generated <n> noise samples
==========================================
```

Note: Both tests require Docker containers to be running (set up using `./run_exp.sh setup`). Results and logs are saved to `test_result/`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our two-way marginal computation (joint histogram) protocol achieves at least a four-order-of-magnitude improvement in communication and a five-order-of-magnitude improvement in running time compared with the existing protocol, CaPS. Compared with the alternative method Sort-count, we also make $122\times \sim 1218\times$ improvement in communication and $30\times \sim 290\times$ improvement in running time. This is demonstrated by experiment (E1), with results shown in Figure 8.

(C2): Our secure noise generation protocol (discrete Gaussian and discrete Laplace) achieves significantly smaller running time compared to the previous protocols. Our improvement primarily lies in the communication round, which is suitable for a high-latency network environment. This is demonstrated by experiment (E2), with results shown in Figure 9 and Table 4.

(C3): Our end-to-end distributed synthesis protocol can generate differentially private synthetic datasets with practical efficiency. For example, we can synthesize the "Adult" dataset in 24 minutes in a real-world WAN setting. This is demonstrated by experiment (E3), with results shown in Table 5. Using our discrete noise implementation, we can still achieve a comparable utility (ML accuracy and query error) as the centralized DP data synthesis algorithm. This is demonstrated by experiment (E4), with results shown in Table 6.

A.4.2 Experiments

Most of our experiments are executed on four Docker containers. One acts as the client to generate all the DPF keys and sends these keys to three other containers acting as the three servers. We suggest using the following command to run all the experiments:

```
nohup ./run_exp.sh all > exp.log 2>&1 &
```

It will take about 7 hours to finish all the experiments in our paper. After finishing, all the numerical results will be saved to the path `result/` and all the graphs in our paper will be saved to the path `result_figures/`, which should contain the following files:

```
result_figures/
  Figure8_marginal_cost.pdf
  Figure8_marginal_cost.png
  Figure9_noise_time.pdf
  Figure9_noise_time.png
  Table4_detailed_noise.csv
  Table5_end2end_generation.csv
  Table6_utility_comparison.csv
  Table6_utility_comparison.tex
```

The names of these files correspond to the figure and table numbers in our paper. Next, we describe each experiment in detail and provide an individual command for each experiment.

(E1): Two-Way Marginal Computation [1 human-minute + 2 compute-hours + 32GB disk]: Compare our DPF-based two-way marginal computation protocol against two baselines (Sort-count and CaPs) across varying numbers of records n and domain sizes u. This supports claim (C1), with results shown in Figure 8 in our paper.

How to: Run the marginal comparison script. The script benchmarks three methods (Our Method, Sort-count,

CaPs) by varying n from 1,000 to 10,000 (with fixed $u = 100$) and varying u from 20 to 200 (with fixed $n = 10{,}000$). For each configuration, it records wall-clock time (seconds) and communication cost (MB). After the benchmarks complete, it automatically generates Figure 8 using the plotting environment.

Preparation: Ensure the Docker containers (`client`, `party0`, `party1`, `party2`) are running. The environments for DPF (via Bazel) and the MP-SPDZ environment have already been configured during installation.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually, use command:

```
./run_exp.sh marginal
```

Results: All the raw CSV data will be saved as:
`result/vary_n_time.csv`,
`result/vary_n_comm.csv`,
`result/vary_u_time.csv`,
`result/vary_u_comm.csv`.
Each CSV contains 3 rows corresponding to our method, `sort_count`, and `caps`. The Figure 8 in our paper is:
`result_figures/Figure8_marginal_cost.pdf`

(E2): Noise Generation Comparison [1 human-minute + 2 compute-hours + 32GB disk]: This experiment evaluates our CDF-based distributed noise generation protocols (CDF-dGauss, CDF-dLap) against the ODO baselines (ODO-dGauss, ODO-dLap) for discrete Gaussian and discrete Laplace distributions. It supports claim (C2), with results shown in Figure 9 and Table 4 in our paper.

How to: The script has two parts: (1) varying the number of samples $n \in \{4, 16, 64, 256, 1024, 4096, 16384\}$ with fixed scale $= 10$, and (2) varying scale $\in \{1, 5, 10, 15, 20, 25, 30\}$ with fixed $n = 1024$. For each configuration and method, it records wall-clock time (seconds), communication cost (MB), and number of rounds. It also runs a detailed comparison with $n = 16384$ and scale $= 30$. After all benchmarks are complete, it generates Figure 9 and Table 4.

Preparation: Ensure the Docker containers (`party0`, `party1`, `party2`) are running. The MP-SPDZ environment (libraries, SSL certificates, IP file) has already been configured during installation.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually:

```
./run_exp.sh noise
```

Results: All the raw CSV data will be saved as
`result/noise_vary_n_time.csv`,
`result/noise_vary_n_comm.csv`,
`result/noise_vary_n_rounds.csv`,
`result/noise_vary_scale_time.csv`,
`result/noise_vary_scale_comm.csv`,
`result/noise_vary_scale_rounds.csv`.
Each CSV contains 4 rows corresponding to CDF-dGauss, CDF-dLap, ODO-dGauss, and ODO-dLap.
The Table 4 data in our paper is saved as:
`result_figures/Table4_detailed_noise.csv`
The Figure 9 in our paper is saved as:
`result_figures/Figure9_noise_time.pdf`

(E3): End-to-End TDS Generation [1 human-minute + 30 compute-minutes + 32GB disk]: Run the distributed AIM synthesis pipeline on the UCI Adult dataset, combining DPF-based marginal computation with general MPC-based selection and measurement. This supports claim (C3), as shown in Table 5 of our paper.

How to: Run the end-to-end generation script. The script first performs DPF marginal computation (keyGen on the client, keyEval on 3 parties), then runs the distributed AIM synthesis (`dist_aim_mpc.py`) with 35 AIM rounds, $\varepsilon = 1.0$, and $\delta = 10^{-9}$. The DPF phase runs first, and its timing is passed to the synthesis phase for combined reporting of per-phase breakdowns (marginal computation, selection, measurement) and total time/communication.

Preparation: Ensure four Docker containers (`client`, `party0`, `party1`, `party2`) are running and the Conda environment `dp_tds_plot` is available. Both have already been configured during installation.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually:

```
./run_exp.sh tds
```

Results: The Table 5 data in our paper is saved as:
`result_figures/Table5_end2end_generation.csv`
This file contains per-phase breakdowns (marginal computation, selection, measurement, generation) and total time/communication.

(E4): Utility Comparison [5 human-minutes + 30 compute-minutes + 32GB disk]: Compare the downstream ML utility and query accuracy of synthetic data generated by standard (centralized) AIM versus our distributed AIM implementation. This supports claim (C3), with results shown in Table 6 in our paper.

How to: Run the utility comparison script. The script runs both standard AIM (continuous Gaussian noise + exponential mechanism) and simulated distributed AIM (discrete Gaussian noise + report-noisy-max) on the Adult dataset for each privacy budget $\varepsilon \in \{0.1, 0.5, 1.0, 5.0, 10.0\}$. For each run, it trains downstream ML classifiers (logistic regression, decision tree, random forest, MLP, XGBoost, CatBoost) on the generated synthetic data and evaluates on real test data, recording accuracy, F1 score, and AUC. It also computes the

conditional query error between the synthetic and real data distributions.

Preparation: Ensure the `tab_bench` conda environment is available. It has already been configured during installation. This experiment does not require Docker containers.

Execution: If you have run "`nohup ./run_exp.sh all`", then no further action needs to be taken. To run this experiment individually:

./run_exp.sh utility

Results: The Table 6 data in our paper is saved as: `result_figures/Table6_utility_comparison.csv` This file contains per-ε accuracy, F1 score, AUC, and query error for both standard and distributed AIM. A LaTeX formatted table is also generated at: `result_figures/Table6_utility_comparison.tex` Distributed AIM should achieve similar accuracy, F1 score, and query error as standard AIM across all tested privacy budgets.

A.5 Notes on Reusability

Our artifact is designed to be modular and reusable:

Using individual primitives: The two-way marginal computation based on three-party DPF and noise generation protocols can be used independently in other MPC applications. Their source codes are located in `src/3party-dpf/` and `src/libs/disGauss.py` with clear interfaces.

In particular, we present the first implementation of a three-party DPF that supports local multiplication. This is especially helpful to the data analysis in a vertically distributed setup.

Adjusting parameters:

- Network conditions: Use `./docker/setup_env.sh wan [latency_ms] [bandwidth_mbit]` to simulate different network settings
- Other datasets: By modifying the marginal computation protocol in `src/marginal/`, and changing the parameters in `src/marginal/libs/e2e/dist_aim_mpc.py`, the generation protocol can support other datasets.

Extending the protocols: The MPC protocols are implemented using a combination of secret sharing and function secret sharing. Researchers can extend the protocols by:

- Adding new noise generation protocols in `src/noise/`.
- Improving the marginal computation protocol (three-party DPF) in `src/3party-dpf/`, including extending our security model to malicious security or supporting a larger number of parties.
- Integrating with other synthesis algorithms beyond AIM.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: The Adverse Effects of Omitting Records in Differential Privacy: How Sampling and Suppression Degrade the Privacy–Utility Tradeoff

Àlex Miranda-Pascual
Karlsruhe Institute of Technology
Universitat Politècnica de Catalunya
alex.pascual@kit.edu

Javier Parra-Arnau
Universitat Politècnica de Catalunya
javier.parra@upc.edu

Thorsten Strufe
Karlsruhe Institute of Technology
thorsten.strufe@kit.edu

A Artifact Appendix

A.1 Abstract

The artifact contains the code necessary to reproduce all of the experiments and plots presented in the paper. The main folder is divided into six subfolders, each covering a different independent experiment. In addition to the main `README.md` file, each folder contains a specific `README.md` for that experiment. The six experiments are written in Python.

The first four folders cover the main paper evaluation for different mechanisms. Each folder covers the utility evaluation and comparison of a mechanism with and without sampling (Section 3), and with and without outlier-score suppression (Section 6). These folders are:

(E1). `NoisyAverage`, which covers the mean computation with NoisyAverage.

(E2). `ReportNoisyMax`, which covers the mode computation with report noisy max and the exponential mechanism.

(E3). `Clustering-DPLloyd`, which covers the DPLloyd clustering algorithm.

(E4). `Clustering-kmedian`, which covers on the k-median clustering algorithm.

In addition, we have two additional folders:

(E5). `PrivacyBound`, which checks that the empirical results obtained match our theorized values for Theorem 5.4.

(E6). `PrivacyBoundPlots`, which generates the remaining paper plots: Figures 2 and 4 (or 75 of the long version).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our code consists of Python scripts that require only standard packages. We are not aware of any malicious content or risks associated with downloading and using our artifact. Our artifact is unable to collect any information from its users.

Our work uses either synthetic databases or commonly used, publicly available databases derived from US and Irish census data. These databases have all been previously and thoroughly anonymized and are explicitly released for public research, and statistical and educational use. Our artifact does not attempt to deanonymize or reidentify participants or their data. We include the database sources in the `README.md`.

We acknowledge the potential biases that can accompany any empirical evaluation and that our paper conclusions may differ for other scenarios, suppression algorithms, or variants of differential privacy (DP). We include the exact CSV files and plots of the execution presented in the paper since the code contains randomization.

A.2.2 How to access

The artifact is permanently hosted at Zenodo:
https://zenodo.org/records/17977527.

A.2.3 Hardware dependencies

We ran our experiments on a server equipped with an AMD EPYC 7702P 64-Core Processor. We note that approximately 1.7 GB of RAM is sufficient to run our code. Experiments **(E1)** to **(E5)** are parallelized, and the default number of processes corresponds to the number of CPU cores on the user's machine. This number can be modified in the `config.py` file. We ran our experiments with a parallelization with 64 cores.

A.2.4 Software dependencies

We run our code on Ubuntu 24.04. The code is written entirely in Python 3.8.20. The code requires the `sage` package

and its dependencies, as well as `matplotlib`, `numpy`, `pandas`, `scipy`, and `tqdm`.

For our experiments, we work with Conda and its environments. We recommend following their user guide to install Conda: https://docs.conda.io/projects/conda/en/stable/user-guide/index.html. We used the version `conda 24.7.1`.

A.2.5 Benchmarks

The artifact size is around 2.5 GB, as it contains the databases tested and the output from one run of each the experiment. Running the experiments again with the default databases and parameters will not require additional disk space.

A.3 Set-up

A.3.1 Installation

We provide an `environment.yml` file in the main folder of the artifact to easily set up an environment with all the necessary dependencies. After downloading the artifact, one can set up and activate the environment with the following commands:

```
conda env create -f environment.yml
conda activate SamplingAndSuppression
```

A.3.2 Basic Test

After setting up the environment, it is only necessary to run the main Python script in each folder to obtain the results for the variables in the paper. This can be done using the following command:

```
python main.py
```

No additional input parameters are required to obtain the paper results. The expected outputs are as follows:

(E1)–(E4): Each experiment returns a folder containing the CSV files with the numerical statistics and the respective plots of sampling and outlier-score suppression evaluations. Each folder contains a `README.md` file describing the output files and their location.

(E5): Each individual script returns a CSV file containing the computer-found (`DiffEvol`) and hypothesized (`HypValue`) values, and their difference (`Difference` = `DiffEvol` − `HypValue`). The script is designed to output error messages in the terminal when a value would contradict our claimed result.

(E6): The script returns the plots in Figures 2 and 4 (Figures 2 and 75 of the long version).

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims of our paper are as follows (only including claims that require experimentation, not including those that are theoretical or from mathematical theorems):

(C1): We conducted an evaluation on uniform Poisson sampling over classic unbounded approximate DP mechanisms. This evaluation reveals that, for fixed privacy levels, the utility guarantees of the DP mechanism with sampling are worse than those of the mechanism without sampling for the tested databases.

(C2): We empirically show that, even when factoring in the privacy amplification, outlier-score suppression in DP worsens the privacy–utility tradeoff analogously to sampling for our tested mechanisms. Despite both techniques providing unfavorable outcomes, our results show that outlier-score suppression rarely outperforms sampling.

In addition,

(C3): We computationally verify that the value for ε^S given in Theorem 5.4 is correct up to an error of $2 \cdot 10^{-7}$.

A.4.2 Experiments

We group Experiments **(E1)** to **(E4)** together because they all serve to justify Claims **(C1)** and **(C2)**.

(E1): **`NoisyAverage`** [5 human-minutes + 4.33 compute-hours[1] + 1.2 GB disk[2]]: Covers the mean computation with the NoisyAverage mechanism.

(E2): **`ReportNoisyMax`** [5 human-minutes + 10.5 compute-hours[1] + 1.1 GB disk[2]]: Covers the mode computation with report noisy max and the exponential mechanism.

(E3): **`Clustering-DPLloyd`** [5 human-minutes + 46 compute-hours[1] (slightly less than 2 days) + 50 MB disk[2]]: Covers the DPLloyd clustering algorithm.

(E4): **`Clustering-kmedian`** [5 human-minutes + 24 compute-hours[1] + 3.5 MB disk[2]]: Covers the k-median clustering algorithm.

These four experiments cover our entire evaluation of sampling (related to Claim **(C1)**) and outlier-score suppression (related to Claim **(C2)**). In these experiments, both the sampling and outlier-score suppression counterparts are run simultaneously because the former is a case of the latter.

Preparation: For each experiment, navigate to the corresponding folder and set up the environment[3].

Execution: For each experiment, run the `main.py` file.

Results: Each of these experiments generates a folder for each tested database. Each folder contains a subfolder with the CSV files (containing the numerical values from the experiment) and the plots. Each of these subfolders includes one or more additional subfolders with the names of the tested columns. Note that these

[1] The runtimes take into account our parallelization with 64 cores.

[2] These values correspond to respective experiment disk space. Note that we provide the output of a full run in the artifact and, therefore, running the experiment again will not require any (significant) additional disk space.

[3] All experiments can be run with the same environment, which is also contained in the main folder for convenience.

Expt.	Claim	File names
(E1)	**(C1)**	`[column]_uniform_Poisson_sampling_[laplace/gaussian]_MPE+Sd.pdf`
(E1)	**(C2)**	`[column]_eps=[epsilon]_delta=[delta]_difference_[laplace/gaussian]_M_minus_MoSChangeEpsDelta_MPE_10--90.pdf`
(E2)	**(C1)**	`[column]_uniform_Poisson_sampling_[laplace/gaussian/exponential/exponential_mechanism]_EmpProb+Sd.pdf`
(E2)	**(C2)**	`[column]_eps=[epsilon]_delta=[delta]_difference_[laplace/gaussian/exponential/exponential_mechanism]_error_M_minus_MoSChangeEpsDelta_10--90.pdf`
(E3)	**(C1)**	`[columns]_uniform_Poisson_sampling_Average+SD.pdf`
(E3)	**(C2)**	`eps=[epsilon]_difference_error_M_minus_MoSChangeEpsDelta_Average_10--90.pdf`
(E4)	**(C1)**	`[columns]_uniform_Poisson_sampling_Average+SD.pdf`
(E4)	**(C2)**	`eps=[epsilon]_difference_error_M_minus_MoSChangeEpsDelta_Average_10--90.pdf`

Table 1: Name of plots that verify the claims **(C1)** and **(C2)** for the experiments **(E1)**, **(E2)**, **(E3)**, and **(E4)**.

folders contain more CSV files and plots than are shown in the paper, covering additional data that may interest some readers. For more details on these plots, refer to the `README.md` file. The names of the specific plots used in the paper to support claims **(C1)** and **(C2)** are shown in Table 1. For convenience, the script also provides copies of only those relevant plots into the `PaperPlots` folders.

The sampling plots (those with the **(C1)** tag in Table 1) show the utility values of the mechanism with and without sampling under the same privacy guarantees. Clearly, the mechanism with sampling provides worse utility for almost all sampling rates, as seen is evident across all plots[4]. This result confirms Claim **(C1)** for our study.

The outlier-score suppression plots (those with the **(C2)** tag in Table 1) directly show the difference in utility values between the mechanism with and without outlier-score suppression. A negative value indicates that the mechanism utility with outlier-score suppression is worse than that without it. All of our plots show negative values for nearly all points[4], which confirms Claim **(C2)**. In addition, we can see by that the diagonal, representing sampling, usually has smaller absolute values than other nearby values (more precisely, when compared to those values with the same proportion of deleted records). This suggests that outlier-score suppression rarely outperforms sampling for our tested mechanisms and databases.

In addition, the `ViewPaperPlots.html` file in the main folder allows the reader to easily find and open all the figures shown in the paper and its long version.

[4]The modes of the `hours-per-week` column of the Adult database and the `HighestEducationCompleted` column of the Irish database correspond to around half of the records in each database. Therefore, the modes are thus extremely robust to noise, and the respective DP mechanisms (with and without sampling and outlier-score suppression) return no empirical error. Nevertheless, we still included the plots for completeness. Please refer to the paper for more details.

(E5): `PrivacyBound` [1 human-minute + 17 compute-hours[1] + 200 MB disk[2]]: This experiment verifies that the ε^S expression we give in Theorem 5.4 is correct (up to an error of $2 \cdot 10^{-7}$) for the wide range of values of ε, m, and M as discussed in the paper. More precisely, our proof of Theorem 5.4 requires computational assistance to find the maximum of two functions over a domain defined by four or five continuous variables. We use this experiment to check that our theorized maximum is indeed the actual maximum by using optimization functions of `sage`. Please see the exact functions and more details in Remark 11 of the paper, and Remarks 22 and 24 of the long version.

Preparation: Redirect to the `PrivacyBound` folder and set up the environment[3].

Execution: Run the `main.py` file.

Results: Two CSV files are generated, one for each function, each containing the empirical value obtained by the computational optimization (`DiffEvol`), our hypothesized value given by our formula (`HypValue`), and the difference (`Difference` = `DiffEvol` − `HypValue`). In addition, the terminal outputs an error message if `DiffEvol` is larger than `HypValue` (up to some floating error), which would contradict our result that `HypValue` is the true maximum. Additional error messages are also output if the maximum is obtained in an unexpected case, which would also contradict our claim. Our experiment execution did not provide any errors, supporting Claim **(C3)**. Since our hypothesized value is within the maximization domain, we are only interested in measuring when the computational value exceeds our hypothesized value. In this case, the error of less than $2 \cdot 10^{-7}$ (this value is displayed in the terminal at the end of the execution).

(E6): `PrivacyBoundsPlot` [1 human-minute + 10 compute-seconds + 600 KB disk[2]]: This experiment gen-

erates Figures 2 and 4 in the paper (Figures 2 and 75 in the long version).
Preparation: Redirect to the `PrivacyBoundsPlot` folder and set up the environment[3].
Execution: Run the `main.py` file.
Results: The outputs are the plots used in the aforementioned figures. We included this code to confirm that our plots precisely describe the values given by our formulas.

A.5 Notes on Reusability

Experiments **(E1)** to **(E4)** can be run with different parameters and variables (e.g., ε, δ, number of iterations, number of clusters). To do so it suffices add arguments to the execution command. Please refer to the corresponding `README.md` file for the precise arguments necessary, which vary by experiments. Different databases can also be tested as long as they are saved as CSV files with labeled columns (i.e., in the same format as our tested databases).

The evaluations on sampling and outlier-score suppression can be run for new mechanisms, but most files in the folder must be redefined and adapted to these mechanisms. Certain files, such as `suppression_privacy_parameters.py`, should not vary between mechanisms.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: High-Accuracy, Poisoning-Resilient Frequency Estimation in the Shuffle Model

Shaoqiang Wu[†], Jingyu Jia[†‡], Yikuan Zhu[†], Xinhao Li[†], Changyu Dong[§], Zheli Liu[†]

[†]CS&CCS, DISSec, AAIS, Nankai University, Tianjin, China

[‡]Automotive New Technology Research Institute of BYD, Shenzhen, China

[§]Guangzhou University, Guangzhou, China

A Artifact Appendix

A.1 Abstract

The artifact is a self-contained Python implementation of the paper's two histogram frequency-estimation protocols, $\mathcal{P}^{\mathrm{H1}}$ (small-domain) and $\mathcal{P}^{\mathrm{H2}}$ (large-domain, compressed), both built on the paper's symmetric binomial-sum noise design. It also includes the five baselines we compare against (LWY, CSUZZ, BC, CZ, GKMP), the aggregated input histograms, and three scripts that regenerate the data behind every empirical result (Table 2, Table 3, Figure 2, and the Appendix E.1/E.2 supplements that appear only in the extended version[1]). All randomness is explicitly seeded, and the result CSVs in the artifact are exactly the ones used to produce the paper's tables and figures. A `--quick` flag on each script provides a functional check that runs in seconds to about a minute; the unflagged scripts perform the full reproduction. The artifact runs on a single CPU core, with no GPU, no specialized hardware, and no network access.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact is non-destructive. It runs entirely in user space, requires no root privileges, makes no network connections, and modifies nothing outside its own `results/` directory (it only reads the provided input histograms and writes result CSVs). It contains no record-level personal data, since every dataset is used only as an aggregated count histogram. The AOL query log is a standard large-domain benchmark in this literature, and also a sensitive case whose 2006 raw release was retracted, so the artifact contains only a 3-character query-prefix histogram over 2^{24} keys, with no raw queries, user identifiers, or timestamps (see `data/README.md`). The poisoning module is a controlled-benchmark simulator used only to measure robustness; it is not a tool for attacking a deployed system.

A.2.2 How to access

The artifact is permanently archived on Zenodo under the concept DOI `10.5281/zenodo.20405418`, which always resolves to the latest version.

A.2.3 Hardware dependencies

None beyond a standard CPU. Any x86-64 machine suffices; the workloads are single-threaded with a resident-memory footprint of at most 4 GB. No GPU, accelerators, or specialized hardware are required.

A.2.4 Software dependencies

Python 3.11–3.14 (validated on 3.14) and the two scientific-computing packages pinned in `requirements.txt`, `numpy==2.4.4` and `scipy==1.17.1`. Both provide prebuilt binary wheels for CPython 3.11–3.14, so installation succeeds on a clean Ubuntu 24.04 LTS (Python 3.12) or Windows 11 machine through `pip` alone, without a compiler. Any `numpy` $\geq$ 1.26 with `scipy` $\geq$ 1.11 reproduces the reported numbers within the tolerance documented in the README.

A.2.5 Benchmarks

All input data is included; nothing is downloaded at run time. The four aggregated histograms in `data/` are `city.csv` (d=40, $n\approx$156k), `fire.csv` (d=272, $n\approx$681k), `occupation.csv` (d=529, $n\approx$123k), and `aol_3char_hist.npz` (d=2^{24}, subsampled to n=10^5). Provenance and licensing are documented in `data/README.md`.

[1]The extended version (https://eprint.iacr.org/2026/1218) reports MAE±SD and bit-level communication cost in its Appendix E.1, and the full poisoning grid in its Appendix E.2 (Figure 3); both are finer-grained views of the Table 2 and Figure 2 data, produced from the same result CSVs. The camera-ready version omits these two empirical supplements; its appendices contain the two protocol proofs and the anonymous message-control mechanism.

A.3 Set-up

A.3.1 Installation

From the artifact root, create a virtual environment and install the two dependencies:

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

On Windows, activate the environment with `venv\Scripts\activate` instead of the `source` line. No `PYTHONPATH` export is needed, because each script in `reproduce/` adds the artifact root to `sys.path` itself.

A.3.2 Basic Test

Run a quick functional check that exercises the full protocol pipeline (preprocessing, local randomizer, shuffler, analyzer) and all baselines:

```
python reproduce/table2.py --quick
```

This prints a six-row table for City at ε=1.0 and writes `results/quick/table2_quick.csv`. Each row reports, for one protocol, the mean absolute error (MAE), the 95th-percentile error, and the messages per user. The run is functioning correctly if it shows the expected qualitative pattern. Specifically, the two low-message protocols are Ours (≈ 1.5 messages per user) and LWY (≈ 1.0); in MAE, Ours $\leq$ LWY, while CSUZZ, BC, and CZ are several-fold higher; and GKMP reaches low error only at a very large message count (~28,700 messages per user for City).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): On small-domain categorical data, $\mathcal{P}^{\mathrm{H1}}$'s MAE is essentially equal to that of the strongest bounded-message baseline (LWY) at ε=0.25 and lower for $\varepsilon \geq 0.5$, by up to nearly 2×, at comparable messages per user, and far below that of CSUZZ, BC, and CZ. GKMP reaches low error only with thousands to millions of messages per user, far above the per-user message limit this work targets. Proven by (E1); reported in paper Table 2 and Appendix E.1.

(C2): On the AOL large-domain workload, the compressed protocol $\mathcal{P}^{\mathrm{H2}}$ matches LWY's accuracy at the same hashed dimension d_h and comparable per-user communication. Proven by (E2); reported in paper Table 3.

(C3): The poisoning influence of $\mathcal{P}^{\mathrm{H1}}$ grows roughly linearly in the corrupted-user ratio m/n (up to 30%) and stays far below that of CSUZZ, BC, and CZ. Proven by (E3); reported in paper Figure 2 and Appendix E.2 (Figure 3).

A.4.2 Experiments

The result CSV shipped for each experiment holds the exact values behind the paper's tables and figures. Each full run is a single, unattended command that recomputes that CSV from scratch, overwriting it. The recomputed numbers reproduce the paper's tables and figures bit-for-bit on the pinned NumPy/SciPy versions (the CSUZZ and CZ columns of Table 2 to within one or two in the last printed digit, which affects no comparison in the paper), and within about 0.1% on MAE on any other `numpy` $\geq$ 1.26 (see the README). The `--quick` run instead writes to `results/quick/` and leaves the shipped CSVs untouched.

(E1): *Small-domain MAE* [≈5 human-minutes + 30–60 compute-minutes full, or ~5 s with `--quick`; <1 GB disk].
How to: Reproduce paper Table 2 (Claim C1). The full run evaluates all six protocols across the three small-domain datasets and six privacy levels.
Preparation: Complete the Set-up above; no per-experiment preparation is needed.
Execution: `python reproduce/table2.py` (full), or with `--quick`.
Results: Output `results/table2_with_p95.csv`. The MAE of Ours is at or below that of LWY for $\varepsilon \geq 0.5$ and essentially equal at ε=0.25, at comparable messages per user, and well below CSUZZ/BC/CZ. E.g., for City at ε=1.0, Ours 5.42 and LWY 5.96.

(E2): *AOL matched-d_h accuracy* [≈5 human-minutes + 2–3 compute-hours full, or ~30 s with `--quick`; <1 GB disk].
How to: Reproduce paper Table 3 (Claim C2). The full run compares $\mathcal{P}^{\mathrm{H2}}$ with LWY's P_{FE1} on the AOL histogram at two matched hashed dimensions d_h across six privacy levels.
Preparation: None beyond the Set-up.
Execution: `python reproduce/table3.py` (full), or with `--quick`.
Results: Output `results/aol_matched_b.csv`. `our_mae` ≈ `lwy_mae` and `our_msgs` ≈ `lwy_msgs` in each row at both d_h values. E.g., at d_h=65 and ε=1.0, Ours ≈ LWY ≈ 32.5.

(E3): *Poisoning influence* [≈5 human-minutes + ~5 compute-hours full, or ~1 min with `--quick`; <1 GB disk].
How to: Reproduce paper Figure 2 and Appendix E.2 (Claim C3). The full run measures each protocol's poisoning influence on the three small-domain datasets as the corrupted-user fraction m/n sweeps from 0.1% to 30%.
Preparation: None beyond the Set-up.
Execution: `python reproduce/figure2.py` (full), or with `--quick`.
Results: Output `results/poisoning_baselines.csv`. The influence of Ours and LWY stays far below CSUZZ/BC/CZ at every m/n, by roughly one to nearly

three orders of magnitude depending on the dataset and ε. E.g., for Fire at ε=1.0 and m/n=30%, Ours $\approx$ 1.0, LWY $\approx$ 0.87, CSUZZ $\approx$ 82.9, BC and CZ $\approx$ 163.

A.5 Notes on Reusability

Each `protocols/*.py` module is self-contained and can be imported independently of the runners. The modules are `small_domain.py` ($\mathcal{P}^{\mathrm{H1}}$), `large_domain.py` ($\mathcal{P}^{\mathrm{H2}}$ and LWY's P_{FE1}), `baselines.py` (the small-domain baselines), and `poisoning.py` (the poisoning simulator). New datasets are supplied as count histograms (one integer per category per line for the CSV format). The implementation approach, determinism, the per-script RNG seeds, and the cross-NumPy-version reproduction tolerance are documented in the top-level `README.md`.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Interpolation-Based Optimization for Enforcing ℓ_p-Norm Metric Differential Privacy in Continuous and Fine-Grained Domains

Chenxi Qiu
University of North Texas

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper "Interpolation-Based Optimization for Enforcing ℓ_p-Norm Metric Differential Privacy in Continuous and Fine-Grained Domains" and provides the complete MATLAB implementation for the proposed *anchor-interpolated privacy optimization (AIPO)* framework.

The package contains end-to-end driver scripts to reproduce the paper's main results (e.g., `main_2norm.m`), core algorithmic routines under `./functions/` (including optimization, log-convex interpolation, and violation evaluation), and implementations of representative baselines. It also includes processed OpenStreetMap road-network datasets (nodes/edges) for Rome, London, and New York City under `./datasets/`, plus cached intermediate files under `./intermediate/`. Running the provided scripts generates metrics such as utility loss, mDP violation ratio, and runtime, and writes structured outputs to subfolders under `./results/`. The artifact is released as a single ZIP archive on Zenodo (DOI: `10.5281/zenodo.17851733`).

A.2 Description & Requirements

The artifact is distributed as a single ZIP archive. After extracting, the top-level directory contains:

- Primary driver scripts to reproduce tables/figures:
  `main_2norm.m`,
  `main_1norm.m`,
  `main_granularity.m`,
  `main_ablation_privacybudget.m`
- `functions/`: implementation of anchor-based optimization, log-convex interpolation, evaluation (utility/violation), and baseline methods.
- `datasets/`: processed road-network datasets (`nodes.csv`, `edges.csv`) for each city benchmark.
- `intermediate/`: cached intermediate files (e.g., precomputed loss matrices) to reduce preprocessing overhead.
- `results/`: output directory populated by the scripts (cost/violation/time/PPR and related logs).

Configuration is centralized in `parameters.m` (e.g., number of repeats). Input data and/or download instructions are under `dataset/`. A paper copy (`paper.pdf`) is included for reference. To run the experiments, start MATLAB in the repository root, modify `parameters.m` if needed, and then execute `main_2norm.m` to reproduce the main results. To reproduce the supplementary (appendix) results, run `main_1norm_appendix.m`, `main_ablation_budget_appendix.m`, and `main_granularity_appendix.m`. This organization allows evaluators to quickly locate the README, source code, datasets, and run scripts without navigating extraneous files.

A.2.1 Security, privacy, and ethical concerns

Executing this artifact poses *low risk* to evaluator machine security and data privacy. The MATLAB scripts are intended to run locally in user space, require no administrator/root privileges, and do not ask evaluators to disable security mechanisms. The code writes outputs only to local folders (i.e., `./results/`) and is not designed to modify system files. The included datasets are road-network graphs (nodes/edges) and do not contain personal or sensitive information, and the artifact does not require uploading data or sending telemetry.

A.2.2 How to access

The artifact is archived on Zenodo. It can be downloaded from:

`https://doi.org/10.5281/zenodo.17851733`

(DOI: `10.5281/zenodo.17851733`). The record provides `AIPO_artifact.zip`, which contains the full MATLAB code, datasets, and scripts needed to reproduce the results.

A.2.3 Hardware dependencies

- *CPU.* A commodity x86_64 machine; a multi-core processor is recommended (4+ cores).

- *Memory. 32 GB RAM* is recommended to run
  `main_2norm.m`,
  `main_1norm.m`,
  `main_granularity.m`,
  `main_ablation_privacybudget.m`
  More memory can reduce runtime by avoiding swapping.
- *Accelerators.* Not required (no GPU/FPGA needed).
- *Network.* Not required for normal use.

A.2.4 Software dependencies

- *OS.* Windows 10/11, macOS, or Linux (any recent 64-bit distribution).
- *MATLAB.* MATLAB (R2022b or newer recommended).
- *Toolboxes.* The **Optimization Toolbox** is required for linear programming routines used by some baselines. The **Statistics and Machine Learning Toolbox** is used for random sampling (e.g., `randsample`).
- *Other dependencies.* No external solvers, GPUs, or compilers are required for the provided workflows. Network access is not needed for normal use.

A.2.5 Benchmarks

Included datasets. The artifact bundles all data needed to run the experiments entirely offline. The `datasets/` directory contains preprocessed road-network graphs for: **Rome**, **New York City**, and **London**.

Contents and format. Each dataset provides a road graph within the target region (node/edge lists; `nodes.csv` and `edges.csv`). The provided driver scripts (e.g., `main_2norm.m`) load these files automatically.

Compared methods included. Implementations of the paper's compared mechanisms are included in the artifact alongside AIPO (e.g., exponential-mechanism and LP-based baselines), exposed through the run scripts and the `functions/` directory. No external repositories or additional datasets/models are required.

No external downloads. All datasets are included with the package; internet access is not needed to reproduce results.

Data use and privacy. The datasets are public/aggregated research data and contain no personally identifiable information.

A.3 Set-up

A.3.1 Installation

Step 1: Download the artifact. Download the ZIP archive from the Zenodo DOI https://doi.org/10.5281/zenodo.17851733 and extract it to a local folder.

Step 2: Install dependencies. Install MATLAB (R2022b or newer recommended) with the following toolboxes: (i) **Optimization Toolbox** (required for LP-based baselines), and (ii) **Statistics and Machine Learning Toolbox** (used for `randsample`).

A.3.2 Basic Test

This test verifies that (i) MATLAB can locate and execute the artifact code, (ii) datasets can be loaded, (iii) the core optimization/interpolation pipeline runs end-to-end, and (iv) result files are generated.

Prerequisites. MATLAB is launched in the artifact root directory and the code is on the path: `addpath(genpath(pwd));`

Command. Run the MATLAB file `main_2norm.m`. No input parameters are required (the script uses the default settings shipped with the artifact).

Expected successful output. The test is successful if:

- MATLAB finishes without errors, and
- the `results/2norm/` directory is updated with output files produced by the run.

Troubleshooting. If MATLAB reports missing functions (e.g., `randsample` or LP solvers), verify that the required toolboxes are installed and that `addpath(genpath(pwd))` was executed in the artifact root directory.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *AIPO empirically satisfies* (ε, d_p)*-mDP on the evaluated city benchmarks, while discretization-based baselines (e.g., LP and COPT) may exhibit nonzero empirical violations.* Supported by (E1) / Table 1.

(C2): *Under the same privacy budgets, AIPO improves utility loss compared to predefined-noise mechanisms (Laplace, EM, TEM) and hybrid methods (COPT, RMP).* Supported by (E2) / Table 2.

(C3): *AIPO is more time-efficient than optimization-based baselines (LP, COPT).* Supported by (E3) / Table 3.

A.4.2 Experiments

All three experiments (E1)–(E3) are executed together by running the MATLAB script `main_2norm.m`. With the default setting (`NT_TEST`=1), each dataset run typically takes about 5–6 hours, so running all three datasets takes about 15–18

hours (runtime may vary by machine). Increasing `NT_TEST` scales the runtime approximately linearly.

Preparation: Extract the artifact ZIP, start MATLAB in the artifact root, and run:

```
addpath(genpath(pwd));
```

To set the number of repeats, set value of "`NT_TEST`" in `parameters.m`.

Execution: Run:

```
main_2norm.m
```

(E1): *(Supports C1) mDP violation evaluation.* MATLAB prints a table titled: "`[city] road map - Violation ratio`", as Fig. 1 shows:

```
================================================
rome road map - Violation ratio
================================================
Method        e=0.2   e=0.4   e=0.6   e=0.8   e=1.0   e=1.2   e=1.4   e=1.6
------------------------------------------------------------------------------
Pre-defined Noise Distribution:
EM          0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000
Laplace     0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000
TEM         0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000

Hybrid Method:
RMP         0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000
COPT        1.978±0.000  8.097±0.000  6.294±0.000  5.998±0.000  5.723±0.000  5.476±0.000  5.165±0.000  4.929±0.000

LP          2.076±0.000  2.108±0.000  1.817±0.000  1.977±0.000  1.652±0.000  1.362±0.000  1.017±0.000  0.834±0.000
AIPO-R      9.285±0.000  7.582±0.000  5.012±0.000  3.314±0.000  2.099±0.000  1.250±0.000  0.898±0.000  0.573±0.000
AIPO*       0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000  0.000±0.000
```

Figure 1: Printed table for mDP violation ratio (Rome roadmap).

The results are also saved to:

```
results/violation/city/violation_method.mat
```

Verify that AIPO and predefined-noise mechanisms (EM, Laplace, TEM) to achieve 0% empirical violations, while relaxed/approximate baselines (COPT, LP, AIPO-R) may yield nonzero violation ratios, matching the results in Table 1.

(E2): *(Supports C2) Utility-loss comparison.* MATLAB prints a table titled: "`[city] road map - Utility loss`", as Fig. 2 shows:

```
================================================
rome road map - Utility loss
================================================
Method        e=0.2   e=0.4   e=0.6   e=0.8   e=1.0   e=1.2   e=1.4   e=1.6
------------------------------------------------------------------------------
Pre-defined Noise Distribution:
EM          8.54±0.00   8.51±0.00   8.50±0.00   8.49±0.00   8.49±0.00   8.49±0.00   8.49±0.00   8.48±0.00
Laplace     8.54±0.00   8.51±0.00   8.50±0.00   8.49±0.00   8.49±0.00   8.49±0.00   8.49±0.00   8.48±0.00
TEM         8.59±0.00   8.59±0.00   8.59±0.00   8.59±0.00   8.59±0.00   8.59±0.00   8.59±0.00   8.59±0.00

Hybrid Method:
RMP         5.86±0.00   5.19±0.00   5.00±0.00   4.92±0.00   4.88±0.00   4.85±0.00   4.83±0.00   4.82±0.00
COPT        7.71±0.00   8.51±0.00   8.48±0.00   8.47±0.00   8.46±0.00   8.44±0.00   8.41±0.00   8.37±0.00

LP          4.89±0.00   3.18±0.00   2.56±0.00   2.26±0.00   2.11±0.00   2.03±0.00   1.99±0.00   1.97±0.00
AIPO-R      3.93±0.00   2.99±0.00   2.62±0.00   2.46±0.00   2.36±0.00   2.32±0.00   2.29±0.00   2.27±0.00
LB          2.32±0.00   1.78±0.00   1.74±0.00   1.73±0.00   1.73±0.00   1.73±0.00   1.73±0.00   1.73±0.00
AIPO*       5.64±0.00   4.56±0.00   3.90±0.00   3.47±0.00   3.18±0.00   2.96±0.00   2.82±0.00   2.71±0.00
```

Figure 2: Printed table for utility loss (Rome roadmap).

The results are also saved to:

```
results/cost/city/loss_method.mat
```

Verify that the relative ordering among methods matches Table 2: AIPO achieves lower utility loss than predefined-noise mechanisms (Laplace, EM, TEM) and hybrid methods (COPT, RMP), while incurring higher utility loss than the lower bound (LB) and relaxed/discretization-based methods (AIPO-R, LP).

(E3): *(Supports C3) Computation time evaluation.* MATLAB prints a table titled: "`[city] road map - Computation time (s)`", as Fig. 3 shows:

```
================================================
rome road map - Computation time (s)
================================================
Method        e=0.2   e=0.4   e=0.6   e=0.8   e=1.0   e=1.2   e=1.4   e=1.6
------------------------------------------------------------------------------
COPT        145.972±0.000  152.042±0.000  143.868±0.000  143.131±0.000  143.390±0.000  141.733±0.000  143.319±0.000  142.019±0.000
LP          237.313±0.000   46.383±0.000  262.018±0.000  118.086±0.000   89.109±0.000   65.737±0.000   82.725±0.000  120.175±0.000
AIPO*        15.454±0.000   15.711±0.000   14.776±0.000   13.347±0.000   12.795±0.000   12.496±0.000   12.339±0.000   12.511±0.000
```

Figure 3: Printed table for computation time (Rome roadmap).

The results are also saved to:

```
results/time/city/time_method.mat
```

Verify that AIPO is faster than LP and COPT as in Table 3 (absolute runtimes may vary by machine, but the ordering should be consistent).

A.5 Notes on Reusability

Scaling experiments up/down. The provided run scripts execute predefined parameter sweeps. To *scale down* for quick testing, reduce the number of privacy budgets (e.g., fewer ε values), restrict to a single dataset (e.g., Rome), or disable expensive LP-based baselines. To *scale up*, extend the parameter lists or enable additional baselines.

Running on a new road network. To evaluate on a new region, add a new folder under `datasets/` with `nodes.csv` and `edges.csv` in the same format as the included benchmarks.

Changing the metric or utility model. The code separates dataset loading, metric construction, and utility/loss computation. Users can replace or modify the distance computation (e.g., switch from Euclidean distance to Manhattan distance, or plug in a custom metric) and update the loss-matrix generation accordingly. Cached inputs under `intermediate/` can be deleted to force regeneration when changing these components.

Extending baselines. Baseline mechanisms are organized under `functions/` (including a `benchmarks/` subdirectory). New mechanisms can be added by implementing a perturbation-generator function with the same input/output interface as existing baselines and registering it in the run scripts.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Love, Lies, and Language Models: Investigating AI's Role in Romance-Baiting Scams

Gilad Gressel[1], Rahul Pankajakshan[1], Shir Rozenfeld[4], Ling Li[2], Ivan Franceschini[3], Krishnahsree Achuthan[1], and Yisroel Mirsky[4]*

[1]*Center for Cybersecurity Systems & Networks, Amrita Vishwa Vidyapeetham, Amritapuri*

[2]*Ca' Foscari University of Venice*

[3]*University of Melbourne*

[4]*Ben Gurion University of the Negev*

A Artifact Appendix

A.1 Abstract

This artifact provides the code and datasets to reproduce the content filter evaluation experiments (Section 5) presented in the paper. It includes a curated dataset of romance-baiting, e-commerce, and tax scam conversations, along with scripts to evaluate three major content moderation systems: OpenAI Moderation API, Google Perspective API, and Meta Llama Guard 3.

Note on Scope: Due to ethical and safety concerns regarding dual-use risks, this artifact **does not** include the code for the autonomous LLM agents (Section 4) or the data related to the investigation of scam operations (Section 3). This exclusion complies with the open science policy approved for this submission.

A.2 Description & Requirements

The artifact is a self-contained Python project. It requires access to external APIs (OpenAI, Google) and a machine capable of running Llama Guard 3 (8B model) if local evaluation is desired. All necessary datasets are included in the repository.

A.2.1 Security, privacy, and ethical concerns

Evaluators must provide their own API keys for OpenAI and Google Perspective. These keys are stored in a local `.env` file and are not transmitted except to the respective service providers. The evaluation incurs a small cost (approx. $0.50 USD for the full OpenAI evaluation). Llama Guard 3 runs locally.

*Corresponding author: yisroel@bgu.ac.il

A.2.2 How to access

The artifact is publicly available via a Zenodo record, and the source code, scripts, and datasets are hosted in the accompanying GitHub repository.

A.2.3 Hardware dependencies

- **Minimum (API-based evaluation)**: Any machine with internet access.
- **For Llama Guard 3**: A GPU with at least 16GB VRAM (e.g., NVIDIA RTX 6000, A100, or similar) and approximately 20GB of disk space for model weights.

A.2.4 Software dependencies

- **OS**: Linux (Ubuntu 22.04 verified) or Windows/macOS.
- **Python**: Version 3.9 or higher.
- **CUDA**: Version 11.8+ (required only for Llama Guard on GPU).
- **API Keys**: OpenAI API Key and Google Perspective API Key.

A.2.5 Benchmarks

The artifact requires a pre-generated synthetic conversation dataset and access to three moderation systems used for evaluation.

The dataset is provided in the `data/` directory and consists of 1,000 multi-turn conversations, distributed evenly across four classes:

- **Romance-Baiting Scams**: 250 conversations capturing the Hook and Line stages.
- **E-commerce Scams**: 250 conversations.
- **Tax Scams**: 250 conversations.

- **Neutral**: 250 benign conversations.

Each conversation contains multi-turn dialogues generated to reflect the characteristic structure and language of its respective class.

Benchmarking requires access to the following moderation systems:

- OpenAI Moderation API,
- Google Perspective API,
- Meta Llama Guard 3 (8B), executed locally via Hugging Face.

No additional datasets, fine-tuned models, or external workloads are required.

A.3 Set-up

This section details the steps to prepare the Python environment and configure API keys.

A.3.1 Installation

1. Clone the repository and navigate to the root directory.
2. Create a Python virtual environment and install dependencies:

```
# Option 1: Using Conda
$ conda env create -f environment.yml
$ conda activate scam-moderator-eval
# Option 2: Using uv (fast, modern
    dependency resolver)
# If you don't have uv: pip install
    uv
$ uv venv
$ source venv/bin/activate # On Windows:
    venv\Scripts\activate
$ uv pip install -r requirements.txt
# Option 3: Using standard Python
    venv and pip
$ python -m venv
$ source venv/bin/activate # On Windows:
    venv\Scripts\activate
$ pip install -r requirements.txt
```

3. Copy `.env.example` to `.env` and populate it with your API keys (`OPENAI_API_KEY`, `GOOGLE_PERSPECTIVE_API_KEY`, and `HUGGINGFACE_TOKEN`).
4. Make the experiment script executable:

```
$ chmod +x run_experiments.sh
```

A.3.2 Basic Test

Run the evaluation script on a small subset (5 conversations per category) to verify the setup:

```
$ ./run_experiments.sh --quick
```

Expected output: The script should process 20 conversations (5 per category) for each evaluator. Below is an example output for OpenAI (output for Google and Llama Guard will appear similarly):

```
[OpenAI] Results saved to: results/openai_results.csv
[OpenAI] Summary:
[OpenAI] e-commerce: 0/5 flagged (0.0%)
[OpenAI] neutral: 0/5 flagged (0.0%)
[OpenAI] romance_baiting: 1/5 flagged (20.0%)
[OpenAI] tax_scam: 0/5 flagged (0.0%)
[OpenAI] Overall: 1/20 flagged (5.0%)
[OPENAI] Completed successfully
```

A.4 Evaluation workflow

This section outlines the key claims and the experiments required to validate them using Section 5 of the paper.

A.4.1 Major Claims

(C1): *Current content moderation systems (OpenAI Moderation, Google Perspective, Llama Guard 3) fail to reliably detect romance-baiting scams, flagging them at significantly lower rates (near 0%) compared to other scam types (e.g., tax scams at >97%).*

(C2): *Even when romance-baiting scam conversations are flagged by content moderators, the detections are predominantly false positives. They are triggered by benign content rather than actual scam indicators.*

(Scope): *Excluded Claims:* The claims relating to the operational structure of scam compounds (RQ1, RQ2) and the effectiveness of LLM agents in building trust (RQ3, RQ4) are excluded from this artifact evaluation due to ethical constraints preventing the release of the agent code and sensitive interview data.

A.4.2 Experiments

(E1): *Content Filter Effectiveness* [5 human-minutes + 14 compute-hours + $2.50 cost]: This experiment is to reproduce the detection rate comparison of major LLM moderators on different types of text scams. Supports claim C1.

Preparation: Ensure `requirements.txt` is installed and `.env` contains valid API keys. Ensure the `data/` directory contains the evaluation datasets.

Execution: Run the main evaluation script covering all three moderators:

```
$. /run_experiments.sh
```

(Optional) To run individual evaluators:

```
$ ./run_experiments.sh openai
$ ./run_experiments.sh google
$ ./run_experiments.sh llama # Requires
    GPU
```

Results: The scripts generate CSV files in the `results/` directory. Compare the resulting detection rates with the pre-computed results in `expected_results/` and Table 2 of the paper.
Expected Trends:

- Romance-baiting: Low detection rate (<20% for OpenAI, 0-2% for others).
- Tax Scams: High detection rate (esp. Llama Guard >95%).
- E-commerce: Moderate-High detection rate.

(E2): Judge LLM Validation [5 human-minutes + 30 compute-minutes]: After running E1, analyze the flagged content to classify each detection as a true positive (actually harmful) or false positive (benign but incorrectly flagged). This experiment supports claim C2.
Preparation: Ensure (E1) has been completed and result files exist in `results/`. Verify `OPENAI_API_KEY` is set for the judge LLM.
Execution: Run the judge evaluation:

```
$ ./run_experiments.sh judge
```

The judge evaluates all flagged conversations from each moderator's results.
Results: The script outputs judged result files (`*_judged.csv`) with a `judgment` column indicating True Positive or False Positive. The summary shows counts of each category. Compare with Table 2 in the paper (FPR).

A.4.3 Manual Execution & Customization

If you prefer to run the scripts directly without the `run_experiments.sh` wrapper, you can use `evaluation/run_all.py` with the following arguments:

- `-evaluator [openai|google|llama|all]`: Choose which system to evaluate.
- `-subset N`: Process only N conversations per category (useful for quick testing).
- `-data-dir PATH`: Specify a custom data directory.
- `-output-dir PATH`: Specify a custom output directory.

A.5 Notes on Reusability

A.5.1 Synthetic Conversation Dataset

The dataset (`data/`) contains 1000 synthetic multi-turn conversations across four categories: romance-baiting, e-commerce scam, tax scam, and neutral. Each conversation is a JSON file with role/statement pairs. Researchers can use this dataset for:

- Benchmarking other content moderation systems
- Training or fine-tuning scam detection models
- Studying conversational patterns in scam scenarios

A.5.2 Dataset Generation Pipeline

The `dataset_generation/` folder contains our generate-and-judge pipeline for creating synthetic scam conversations. The approach uses an LLM to generate conversations and a separate judge LLM to validate quality. Researchers can:

- Generate new scam scenarios by creating new scenario configuration files
- Modify prompt templates to adjust conversation characteristics
- Adapt the pipeline for generating other types of synthetic dialogue

See `dataset_generation/README.md` for detailed documentation.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: "Do Not Mention This to the User": Detecting and Understanding Malicious Agent Skills in the Wild

Yi Liu[1,*], Zhihao Chen[1,*], Yanjun Zhang[1], Gelei Deng[2],
Yuekang Li[3], Jianting Ning[4,†], Leo Yu Zhang[1,†]

[1]Griffith University, [2]Nanyang Technological University, [3]University of New South Wales,
[4]Zhejiang Key Laboratory of Digital Fashion and Data Governance, Zhejiang Sci-Tech University

A Artifact Appendix

A.1 Abstract

Reviewers can verify the released dataset (98,380 skills, 157 behaviorally-confirmed malicious) and run a small-batch end-to-end reproduction exercising crawl, static scan, sandboxed dynamic execution, behavioral evidence collection, and optional post-hoc Claude Code–based LLM triage (denoted *CC* hereafter). A separate deterministic analysis suite regenerates the paper's quantitative RQ1/RQ2 results from the released CSVs with no crawl, Docker, or LLM.

A.2 Description & Requirements

The artifact has two top-level components: `data/` holds the released benchmark CSVs, and `code/` holds the reproducibility framework and the deterministic analysis suite. Because the agent-skill ecosystem is live and final malicious labels require manual behavioral verification, the framework defaults to a small-batch experiment; environment variables control larger batches.

A.2.1 Security, privacy, and ethical concerns

1. The released CSVs contain only metadata, with URLs to confirmed-malicious payloads redacted. The framework crawls and runs live agent skills in a monitored Docker container, so run it on a disposable machine or VM.

2. The dynamic executor mounts a host Claude Code credential read-only into the container; because the evaluated skill can influence Claude's behavior, prefer a disposable Claude login.

3. The Docker executor enables tracing and packet capture with additional container privileges; it is suitable for controlled artifact evaluation but is not a strong isolation boundary for arbitrary untrusted code. Captured pcaps and runtime outputs may contain prompts, file paths, or local metadata, so evaluators should not run the artifact on private repositories or with private skills.

4. Per the project README, vulnerabilities found during the study were responsibly disclosed to the affected platform vendors.

A.2.2 How to access

The source is on GitHub at https://github.com/protectskills/MaliciousAgentSkillsBench; the permanent Zenodo archive is https://doi.org/10.5281/zenodo.20285751 (concept DOI, latest release). Reproduce from the evaluated release tag rather than the moving default branch. The Docker sandbox image is `ghcr.io/protectskills/claude-skill-sandbox:lite`, with an offline tarball via the `sandbox-lite-v1` GitHub release for restricted networks.

A.2.3 Hardware dependencies

A modern x86-64 Linux machine with ≥ 4 CPU cores, 8 GB RAM, and 20 GB free disk suffices for the default small-batch experiment; Docker is required for dynamic execution. No GPU is needed—the default Nova-tracer recording mode and the prebuilt `lite` image are CPU-only.

A.2.4 Software dependencies

The host requires Python 3.10+, Docker, Git, `curl`, GNU `timeout` (macOS: expose coreutils `gtimeout` as `timeout`), and a local Claude Code CLI login or an Anthropic-compatible API key (both documented in the README); the optional CC triage additionally needs `jq`. Host-side Python dependencies are in `code/requirements.txt`. The default reproduction pulls the prebuilt sandbox image (≈ 300 MB), so the host needs no `strace`, `tcpdump`, or Nova-tracer hooks—they ship inside the image. A SkillsMP API key is required for the default crawl (free at `skillsmp.com`); a GitHub token is optional but recommended for larger downloads.

*Co-first authors. †Co-corresponding authors.

A.2.5 Benchmarks

The released benchmark, from the paper's January 2026 measurement of two registries (skills.rest, skillsmp.com), consists of:

- `data/skills_dataset.csv`: the Tier-1 snapshot of 98,380 skills across 11,246 repositories, with columns `source`, `repo`, `skill_name`, `classification`, `url`. The `classification` labels are mutually exclusive—`safe` (94,093), `suspicious` (4,130), `malicious` (157)—and `suspicious` + `malicious` = 4,287 reconstructs the paper's nested Tier-2 candidate set. The `url` field is redacted for `malicious` rows and rows sharing their repository.
- `data/malicious_skills.csv`: the curated Tier-3 dataset of 157 behaviorally-confirmed malicious skills (21 from skills.rest, 136 from skillsmp.com) across 69 repositories. Beyond the shared schema it carries a `Pattern` column (semicolon-separated vulnerability-pattern labels) and an aligned per-instance `Severity` column (CRITICAL/HIGH/MEDIUM/LOW).

The default reproduction does not download or execute all 98,380 skills; it runs a configurable small-batch over current SkillsMP results—a few repositories downloaded and statically scanned, with one skill dynamically executed and (if `ENABLE_CC_ANALYSIS=true`) a small post-hoc triage queue.

A.3 Set-up

A.3.1 Installation

From `MaliciousAgentSkillsBench/code/`, create a Python 3.10+ virtual environment, run `pip install -r requirements.txt`, then launch `python3 helper.py`. For first-time setup, step through `1. init` (writes `code/.env`), `2. doctor` (host checks), and `3. image` (pulls the prebuilt image; `load-tar` for the offline tarball, `full-cpu`/`full-custom` for local builds). Menu actions `4. run`, `5. status`, `6. clean` run the small-batch experiment, inspect/edit configuration, and prune outputs.

A.3.2 Basic Test

From `code/`, launch `python3 helper.py` and select `2. doctor`. `SKILLSMP_API_KEY` and the Claude credentials must report `OK` before E2 (the SkillsMP crawl is the only entry into the pipeline). Note that `host strace`/`tcpdump` may remain `MISSING` in some configurations.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The released benchmark CSVs load with the documented schema, and the curated Tier-3 dataset contains exactly 157 behaviorally-confirmed malicious skills (21 from skills.rest, 136 from skillsmp.com), matching Table 2 of the paper. This is validated by experiment E1.

(C2): The artifact can run the framework's end-to-end analysis workflow on a bounded sample: crawl, mapping, download, static scan, dynamic execution, and behavioral evidence collection, with optional post-hoc LLM-assisted triage. This is validated by experiment E2.

(C3): Dynamic execution records behavioral evidence needed for manual security analysis, including Claude output, system calls, network traffic, filesystem changes, run metadata, and Nova-tracer session reports. This is validated by experiment E3.

(C4): The released dataset deterministically reproduces the paper's core quantitative results—the attack-technique taxonomy counts, the pattern co-occurrence matrices, and the headline RQ2 hypothesis tests—offline, with no crawl, Docker, or LLM; the paper's RQ3 sophistication and shadow-feature results lie outside the artifact's scope. This is validated by experiment E4.

A.4.2 Experiments

The experiments form two independent tracks. E1 and E4 are deterministic and offline—no credentials, Docker, crawl, or LLM—and reproduce the paper's released-data results; E2 and E3 exercise the live, non-deterministic pipeline on a bounded sample (needing a SkillsMP key, Docker, and a local Claude login or API key). Configuration knobs and batch execution are detailed in the artifact README.

(E1): *Dataset integrity check [10–15 human-minutes].*
How to: From the repository root, run `python3 code/analysis/verify_dataset.py` (stdlib-only), which prints the label counts below; verify the remaining schema and redaction properties by loading the two CSVs (pandas or `csv.DictReader`).
Results: `malicious_skills.csv` contains 157 rows—21 `skills.rest`, 136 `skillsmp.com` (Table 2)—with `Pattern` and `Severity` columns; `skills_dataset.csv` follows the schema `source, repo, skill_name, classification, url` with the mutually exclusive split `safe` 94,093, `suspicious` 4,130, `malicious` 157 (total 98,380), where `suspicious` + `malicious` reconstructs the paper's 4,287 Tier-2 candidates; every malicious row carries a redacted `url`.

(E2): *Default small-batch pipeline [30–60 human-minutes + 30–60 compute-minutes, depending on image-pull, live crawl, and model latency].*
Preparation: Configure `SKILLSMP_API_KEY`, start Docker, log in to Claude Code locally (or configure an Anthropic-compatible API key), and pull (or load) the sandbox image (see Set-up).
Execution: From `code/`, launch `python3 helper.py`

and select `4. run` (the default). The helper lists the six pipeline scripts in order—crawl, mapping, download, static scan, run-queue generation, dynamic execution—and prompts before starting; the optional CC steps are appended only when `ENABLE_CC_ANALYSIS=true`. The crawl-through-scan steps abort on empty input, so on a no-data outcome raise `SKILLSMP_MAX_ITEMS`, `DOWNLOAD_LIMIT`, or `SCAN_LIMIT` via `5. status` → `edit` and re-run. A continuation prompt then runs more pending skills without repeating crawl/download/scan.

Results: Successful execution creates gitignored runtime directories under `code/` (`data/`, `workspace/`, `tasks/`, `logs/`). Dynamic-execution evidence is written under `workspace/dynamic/`, one per-run subdirectory keyed by risk, repo, skill, and run-id (inspected in E3); `scan_results/` appears only when the optional CC steps run. A run that hits `EXEC_TIMEOUT` after a Nova-tracer report is written counts as a captured artifact, not a failure (set `RETRY_TIMEOUT_ARTIFACTS=true` to re-execute it).

(E3): *Dynamic execution evidence inspection [15–30 human-minutes; no additional compute].*

How to: After E2, from `code/`, open the newest run directory under `workspace/dynamic/` (layered by risk, repo, skill, run-id) and inspect its artifacts: the Nova-tracer HTML report in a browser, `network.pcap` in Wireshark or `tshark`.

Results: A successful run contains `claude_output.txt`, `strace.log`, `network.pcap`, `filesystem_changes.json`, `metadata.json`, and a `nova-tracer/` subdirectory with session JSONL traces and an HTML report (`strace.log` requires syscall tracing, which is off by default on macOS hosts, where in-container `ptrace` is unreliable). When `EXEC_REPORT_MODE=true`, the directory also contains a Claude-written `claude_execution_report.md`.

(E4): *Deterministic quantitative reproduction [15–25 human-minutes; no crawl, Docker, or model calls].*

How to: From `code/analysis/`, install its `requirements.txt`, then run `taxonomy_counts.py`, `rq1_landscape.py`, `cooccurrence.py`, and `hypothesis_tests.py` (`code/scripts/ 09–11` wrap the last three). All four read only the released `data/malicious_skills.csv`; `analysis/README.md` maps each script to the paper table or figure it regenerates.

Results: `taxonomy_counts.py` prints the attack-technique taxonomy—632 vulnerability instances across 13 techniques over 157 skills—reproducing Table 4, and `rq1_landscape.py` the Table 5 overview plus the RQ1 vulnerability-density and kill-chain-phase breakdowns. `cooccurrence.py` writes the pattern co-occurrence count, odds-ratio, and conditional-probability matrices and the Figure 6 heatmap, and `hypothesis_tests.py` the Fisher's exact tests with Bonferroni correction and the Mann-Whitney severity test. Outputs are deterministic across runs.

A.5 Notes on Reusability

Larger experiments are configured via `5. status` → `edit`, which updates `code/.env` fields (`SKILLSMP_MAX_ITEMS`, `DOWNLOAD_LIMIT`, `SCAN_LIMIT`, `RUN_QUEUE_LIMIT`, `EXEC_WORKERS`); `6. clean` prunes outputs between iterations. The helper's `exec-dir` batches skills from a local directory without a fresh crawl. Beyond `NOVA_PROFILE=record`, the `scan` and `guard` profiles add Nova warnings or `PreToolUse` blocking. The study's confirmed-malicious skills were removed from the live registries after responsible disclosure, so `code/samples/` ships a defanged copy of one—a calculator hiding a reverse shell, its callback endpoint neutralized to `127.0.0.1`. From `code/`, `samples/run_detection.sh` executes it in the Docker sandbox and `samples/run_cc.sh` runs the CC triage, reproducing an end-to-end detection.

A.6 Threats to Validity

- **Live corpus drift.** The ecosystem changes between runs—repositories disappear, branches move, SkillsMP results vary—so the default crawl will not match the paper's January 2026 snapshot.
- **Skills.rest is unreachable for new crawls.** The registry returns a Cloudflare challenge to headless requests, so the artifact does not re-crawl it; the released CSVs are its only source of skills.rest metadata.
- **Non-deterministic LLM behavior.** Claude Code execution is stochastic: dynamic-execution evidence and optional CC labels are reproducible by category, not byte-for-byte.
- **Manual verification stage.** The 157 "confirmed malicious" labels came from human review of the dynamic evidence. The artifact produces that evidence (E2/E3) but does not auto-label, so the count cannot be mechanically re-derived from a small-batch run.

A.7 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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Exposing Resource-Exhaustion DoS Vulnerabilities with Leak-Oriented Minimum Path Covers

Lige Zhan*
Wuhan University
ligezhan@whu.edu.cn

Yafei He*
Wuhan University
yaf2023@whu.edu.cn

Jiang Ming†
Tulane University
jming@tulane.edu

Guojun Peng*
Wuhan University
guojpeng@whu.edu.cn

Jianming Fu*,‡
Wuhan University
jmfu@whu.edu.cn

A Artifact Appendix

A.1 Abstract

Resource leaks occur when programs fail to properly release scarce GC-unmanaged resources, such as file descriptors, sockets, cursors, and database connections. Repeatedly triggering such leaks may exhaust system resources and cause denial-of-service failures. Prior approaches typically enumerate execution paths to identify resources that are not properly released. However, exceptions may trigger non-local control transfers that bypass resource releases, while exhaustive path enumeration may cause path explosion. To address these limitations, we present LeakHunter, a resource-leak detector for Java programs.

The artifact contains the LeakHunter source code, the DroidLeaks and JLeaks evaluation datasets, a runnable functional validation demo, and a pre-built Docker image.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

LeakHunter performs static analysis of Java source code. It does not execute the analyzed Java programs, exploit vulnerabilities, disable security mechanisms, or perform destructive operations. The public artifact is intended for defensive analysis and does not include automated exploitation or DoS triggers, nor does it provide instructions for attacking third-party systems. Any sensitive details concerning real-world issues are released only after coordinated disclosure and remediation, where applicable.

We recommend using Docker to reproduce the artifact for the Functional evaluation. When the recommended Docker command is used, the container is automatically removed after execution because the command includes the `--rm` option. For a more detailed discussion of ethical considerations, please refer to our paper.

A.2.2 How to access

The artifact is permanently available from Zenodo at:

https://doi.org/10.5281/zenodo.18476387

Please download the latest version of `LeakHunter_upload.zip` from the Zenodo record. To prevent accidental downloads of outdated versions, we have restricted access to the older files and left only the latest version publicly accessible. The `LeakHunter_upload.zip` archive contains:

- `Docker/leakhunter-ae-docker.tar`: pre-built AMD64 Docker image for immediate evaluation;
- `source/LeakHunter-source.zip`: source code, the full DroidLeaks and JLeaks evaluation datasets, Dockerfile, dependency descriptions, and documentation;

A.2.3 Hardware dependencies

We recommend an evaluation environment that meets at least the following hardware requirements:

- an x86-64/AMD64 processor;
- at least 8 GB of RAM, with 16 GB recommended;
- at least 30 GB of available disk space.

No GPU, FPGA, specialized processor, hardware performance counter, or other special hardware is required.

A.2.4 Software dependencies

For the recommended evaluation procedure, the evaluator only needs Docker Desktop or Docker Engine. The supplied Docker image already contains:

- Python 3.9;
- OpenJDK 21;
- Graphviz;
- Python packages required by LeakHunter.

For native source execution, the evaluator must install Python 3.9, OpenJDK 21, Graphviz, the packages listed in `requirements.txt`, and `openpyxl`.

A.2.5 Benchmarks

The complete artifact includes the DroidLeaks and JLeaks datasets used in the paper's evaluation. For the Functional badge evaluation, we provide a small subset, `DroidLeaks-refined_demo.csv`, for quick validation. This subset contains five bug-fix pairs, comprising ten analysis inputs. In each pair, the buggy version contains a resource leak, whereas the corresponding fixed version does not. The subset is intended to validate LeakHunter's main detection pipeline and is not intended to reproduce the complete quantitative evaluation reported in the paper.

A.3 Set-up

A.3.1 Installation

We recommend using the pre-built Docker image. After downloading and extracting `LeakHunter_upload.zip`, enter the directory containing `leakhunter-ae-docker.tar` and load the image with:

```
docker load -i leakhunter-ae-docker.tar
```

A successful import should produce output similar to:

```
Loaded image: leakhunter-ae:latest
```

The evaluator may confirm that the image is available by running:

```
docker images leakhunter-ae
```

Alternatively, LeakHunter can be run natively from the source package. From the extracted source directory, install the Python dependencies with:

```
python -m pip install -r requirements.txt
python -m pip install openpyxl
```

A.3.2 Basic Test

We provide a basic functionality test located at `src/scan_basic_test.py`. It can be executed directly using the following command:

```
docker run --rm leakhunter-ae python -u \
src/scan_basic_test.py
```

If the messages `Detection result: LEAK` and `Basic functionality test passed` appear, LeakHunter has been successfully set up and its main detection pipeline is functioning correctly, as shown in Figure 1.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): After inferring resource-management APIs and resource-relevant exceptions, LeakHunter can detect resource leaks. This capability is demonstrated by its successful detection of the leaks in all five buggy methods included in the runnable demo. This claim is validated by experiment (E1), for which the expected results are `TP=5` and `FN=0`.

(C2): After inferring resource-management APIs and resource-relevant exceptions, LeakHunter can correctly recognize non-leaking methods. This capability is demonstrated by its correct classification of all five fixed methods included in the runnable demo. This claim is validated by experiment (E1), for which the expected results are `TN=5` and `FP=0`.

A.4.2 Experiments

(E1): *Runnable Demo Functional Validation* [approximately 2 human-minutes + 10 compute-minutes + 30 GB Docker disk usage].
This experiment validates claims (C1)–(C2) using five bug-fix pairs, comprising ten analysis inputs.
Preparation: Load the supplied Docker image as described in the Installation section:
`docker load -i leakhunter-ae-docker.tar`
Confirm that `leakhunter-ae:latest` is listed by:
`docker images leakhunter-ae`
Confirm that LeakHunter is functioning correctly, as described in the Basic Test section, by running:

```
docker run --rm leakhunter-ae python -u \
src/scan_basic_test.py
```

Execution: Run the functional validation demo:

```
docker run --rm leakhunter-ae:latest python -u \
  runnableDEMO/scan_droidleaks_demo.py
```

Results: For claims (C1) and (C2), inspect the classification of the buggy and fixed versions. Each buggy version should be reported as `LEAK` and counted as a true positive. Each fixed version should be reported as `NO LEAK` and counted as a true negative.
The expected final statistics are:

```
Metrics so far: TP=5, FP=0, FN=0, TN=5,
       Precision=1.0000, Recall=1.0000
```

Figure 2 illustrates one complete output of the demo.

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/usenixsec2026/.

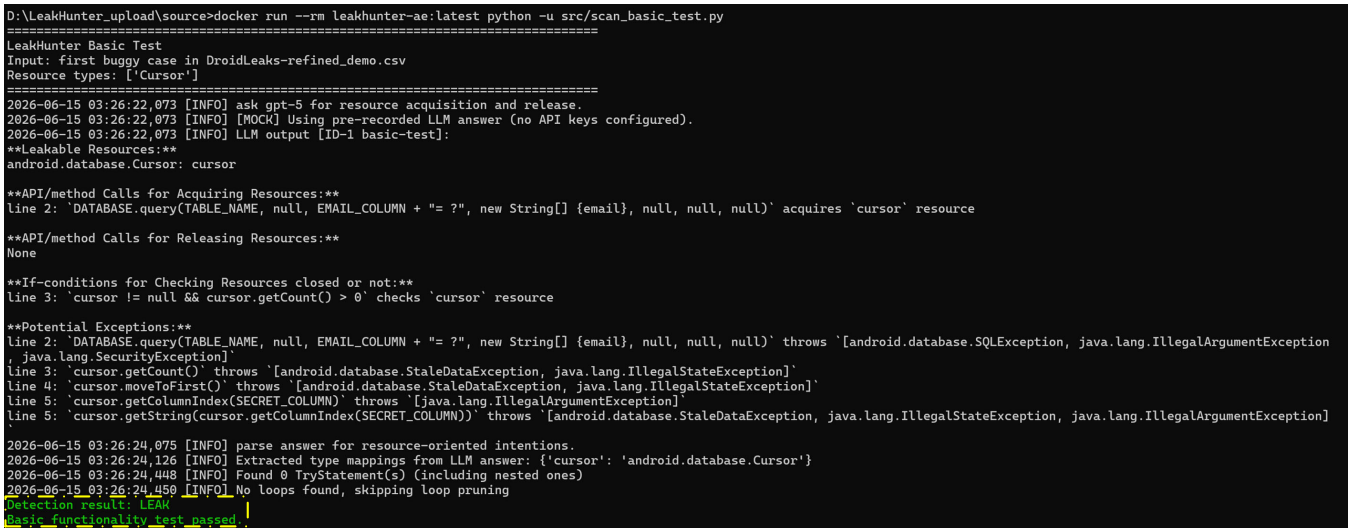


Figure 1: Expected output of the basic functionality test

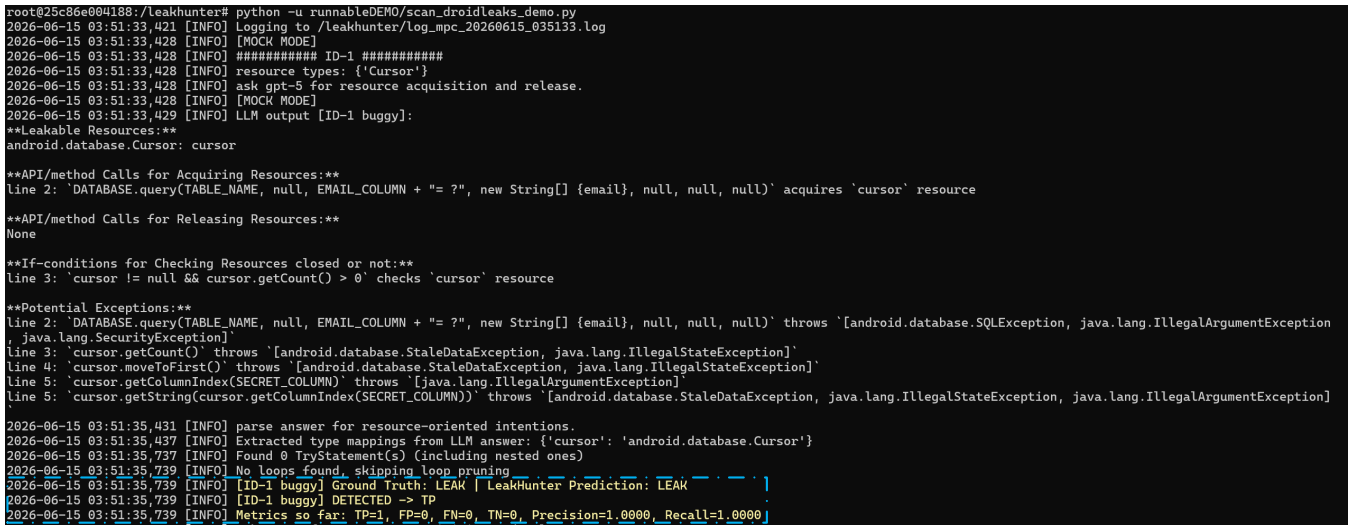


Figure 2: Representative output of the runnable functional validation demo. The output includes the case identifier, identified resource information, leak classification, and cumulative statistics





USENIX Security '26 Artifact Appendix: <United We Defend: Collaborative Membership Inference Defenses in Federated Learning>

Li Bai[1], Junxu Liu[1,2], Sen Zhang[1], Xinwei Zhang[1], Qingqing Ye[1], Haibo Hu[1,2] *
Department of Electrical and Electronic Engineering, The Hong Kong Polytechnic University[1]
PolyU Research Centre for Privacy and Security Technologies in Future Smart Systems[2]

A Artifact Appendix

A.1 Abstract

This artifact contains the core CoFedMID algorithm for defending against membership inference attacks in federated learning, along with detailed documentation for environment setup and demo execution. To facilitate understanding and reproducibility of our experiments, we provide an end-to-end demonstration that covers dataset preparation, federated learning training, and the implementation and evaluation of membership inference attacks, enabling readers to reproduce our results and extend our defense to new datasets or attack settings.

A.2 Description & Requirements

Our experiments were carried out on a machine running CUDA 12.5 with four NVIDIA GeForce RTX 3090 GPUs. The software environment is based on Python 3.9 and relied on several key libraries, including PyTorch, TorchVision, and Torchaudio. All required Python dependencies are listed in the `requirements.txt` file provided in our code repository.

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://zenodo.org/records/17982729.

A.2.3 Hardware dependencies

We conducted our experiments on an Ubuntu system equipped with four NVIDIA GeForce RTX 3090 GPUs, each with 24 GB of memory. The artifact is expected to be compatible with other modern GPU devices.

*Corresponding author: haibo.hu@polyu.edu.hk

A.2.4 Software dependencies

The software environment is based on Python 3.9.25 and relies on the following key libraries: PyTorch 2.6.0, TorchVision 0.21.0, Torchaudio 2.6.0, and Opacus 1.5.4. These specific versions were used in all experiments and are recommended to ensure full compatibility and reproducibility. Additional standard Python dependencies are listed in detail in the `requirements.txt` file.

A.2.5 Benchmarks

Four benchmark datasets are used in our experiments: CIFAR10, CIFAR100, and TinyImageNet. The former two are readily available through PyTorch's built-in dataset utilities. For TinyImageNet, detailed instructions for downloading and preprocessing are provided in the `README.md`. Besides, we focus on two families of image classifier models: ResNet, and WideResNet.

A.3 Set-up

We provide some shell scripts to automate the evaluation.

A.3.1 Installation

We provide a `requirements.txt` file to facilitate environment reproduction and script execution. The necessary dependencies can be installed by running the following commands:

```
conda create -n cofedmid python=3.9 -y
conda activate cofedmid
pip install -r requirements.txt
```

An additional script, `run_verify.sh`, is also provided to test the installation environment.

A.3.2 Basic Test

Basic tests can be run by:

```
bash run_cofedmid.sh
```

For more detailed and comprehensive evaluations and demonstrations, please refer to the `README.md` file for additional information.

A.4 Evaluation workflow

Our experiments mainly rely on two scripts, namely `train_FL.py` and `mia_attack.py`. The former is used to train a federated learning system, while the latter is designed to conduct membership inference attacks in the federated learning setting. Our defense mechanisms are implemented in `train_FL.py` by enabling specific defense options and configuring corresponding hyperparameters. Finally, the training and attack results are saved in CSV format for further analysis.

A.4.1 Major Claims

(C1): CoFedMID achieves utility performance comparable to that of undefended models, as supported by the results of E1 and E2. Further analysis is provided in Section 5.2.

(C2): CoFedMID effectively mitigates membership inference risks in federated learning, as demonstrated by the attack results in E3. For instance, the results reported in Tables 1 and 2 for CIFAR100 with ResNet18 illustrate this effect.

A.4.2 Experiments

We provide a detailed demonstration for reproducing our experiments using CIFAR100 and ResNet18 as an example. Experiments on other datasets and model architectures can be conducted by modifying the corresponding hyperparameters, as described in Section 5.1 of our paper.

(E1): [Undefended Models] We train undefended federated learning models using `train_FL.py`.
How to: Download the required datasets and configure the federated learning setup.
Preparation: For TinyImageNet, run the preprocessing script `utils/preprocess_tiny_imagenet.py`. Other datasets do not require preprocessing.
Execution: Execute `train_FL.py` with the hyperparameters `--defense none` and `--shuffle none`. A demo script `run_baseline.sh` is also provided in our code repositories for convenience.
Results: Collect the training results, which include the training configurations and the achieved test accuracy. For instance, the test accuracy of 0.4669 can be found in `info_cifar100_resnet18.csv`.

(E2): [CoFedMID Implementation] We apply CoFedMID to standard federated learning.
How to: We implement CoFedMID by configuring the federated learning setup to enable CoFedMID defense strategies. The main hyperparameters of CoFedMID include three settings: the number of trusted users (`--trusted_users T`) and the defense strategy (`--defense clsplit-mab-hamp` and `--shuffle layadd`). The former, `--defense clsplit-mab-hamp`, activates the *Class-guided Partition* and *Utility-aware Compensation* components, while the latter, `--shuffle layadd`, enables the *Aggregation-neutral Perturbation* component.
Preparation: Same as E1.
Execution: A demo script `run_cofedmid.sh` is provided for running CoFedMID with all modules enabled under the Pair and Half scenarios.
Results: Similar to E1, the training results are saved in CSV files. For example, the test accuracy of 0.4509 can be found in `info_cifar100_resnet18.csv` for scenarios where a pair of clients collaborate.

(E3): [Attack Simulation] We perform state-of-the-art membership inference attacks in federated learning to evaluate the effectiveness of defense strategies.
How to: Using the intermediate results of local and global models, we run `mia_attack.py` to simulate potential attacks.
Preparation: During E1 or E2, intermediate results were already saved and serve as the attack dataset.
Execution: The attack commands are provided in `run_baseline.sh` and `run_cofedmid.sh` to execute the attacks. Example usage: `python mia_attack.py --dataset $dataset --defense $defense --shuffle $shuffle --folder $folder`
Results: The attack results are saved in CSV files, such as "res_*.csv", including metrics such as MIA, AUC, and TPR@FPR=0.1/0.01/0.001. Results corresponding to CoFedMID can be identified by filtering rows where `defense==clsplit-mab-hamp-layadd`.

In fact, our repository includes detailed demo scripts described in `README.md`, covering training, attack scripts, and result visualization using Jupyter notebooks.

A.5 Notes on Reusability

CoFedMID is designed in a modular manner and can be extended to defend against other privacy attacks in federated learning, such as gradient inversion attacks. In particular, the *Aggregation-neutral Perturbation* module is reusable and can perturb client gradients while preserving the correctness of the global aggregation 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: What the Eyes See, the LLMs Miss: Exploiting Human Perception for Adversarial Text Attacks

Qin Yang[1]*, Lu Malloy[2]*, Joshua Lee[3], Xiaohan Chang[1], Meisam Mohammady[4]
Doowon Kim[2], Yuan Hong[1]

[1]*University of Connecticut*, [2]*University of Tennessee*,
[3]*University of California, Santa Barbara*, [4]*Iowa State University*

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation of Human-Perceptible Adversarial Attacks (HPAA), together with datasets, detector-evaluation pipelines, user-study materials, and analysis scripts required to reproduce the results reported in the paper.

The artifact supports both commercial moderation systems (OpenAI Moderation, Google Perspective API, AWS Bedrock Guardrails, and Azure Content Safety) and local safety models (Llama Guard and ShieldGemma). All experiments were tested on Ubuntu Linux using Python and PyTorch.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact does not execute destructive operations, modify system configurations, or require disabling security mechanisms. The artifact itself does not collect personal information from evaluators. The user-study data included in the artifact have been anonymized and contain no personally identifiable information.

Some evaluation scripts interact with external moderation APIs, including OpenAI Moderation, Google Perspective API, AWS Bedrock Guardrails, and Azure Content Safety. These experiments require Internet connectivity and user-provided API credentials. Requests submitted to these services are subject to the corresponding providers' privacy policies and terms of service.

A.2.2 How to access

The artifact is archived on Zenodo and available at: https://doi.org/10.5281/zenodo.20335299. The source code is also available at: https://github.com/datasec-lab/hpaa.

*Both authors contributed equally to this work.

A.2.3 Hardware dependencies

Most experiments, including attack generation and API-based evaluations, can be executed on a standard workstation. For experiments involving local detector models (e.g., Llama Guard and ShieldGemma), we recommend a GPU with at least 24 GB of GPU memory.

A.2.4 Software dependencies

The artifact has been tested on Ubuntu 22.04 LTS with Python 3.10. All required packages can be installed using the provided `requirements.txt` file.

A.2.5 Benchmarks

The artifact includes the datasets and evaluation resources used in the paper, including datasets for adversarial-example generation, user-study datasets and survey instruments, detector evaluation scripts, configuration files, and analysis utilities.

Evaluations are supported for both commercial moderation APIs (OpenAI Moderation, Google Perspective API, AWS Bedrock Guardrails, and Azure Content Safety) and local safety models (Llama Guard and ShieldGemma). Commercial API evaluations require valid API credentials.

All datasets, configuration files, and analysis scripts required to reproduce the reported experiments are included in the artifact package.

A.3 Set-up

Please confirm that the required software packages are installed on the system as stated above, before proceeding with the following steps.

A.3.1 Installation

Download and extract the artifact package:

```
unzip hpaa_artifact.zip
cd hpaa
```

Install the required dependencies:

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

For experiments involving local detector models (Llama Guard and ShieldGemma), install the optional dependencies:

```
pip install torch transformers accelerate
```

For experiments involving commercial moderation APIs, copy the API template and provide valid credentials:

```
cp .env.example .env
```

Edit `.env` and populate the required API keys.

For evaluations using local detector models, download the corresponding model weights and update the `download_path` field in `src/detectors.yaml` as described in the repository `run.sh` or `README`.

A.3.2 Basic Test

Run the quick single-sample test (Option A in `run.sh`). Successful execution generates the file `./HPAA/optA.M1-W-Hi.csv`. To generate adversarial examples under different settings, uncomment the corresponding options in `run.sh`. To reproduce the configurations evaluated in the paper, adjust -m (M1–M6), -l (W, T, Mix), and -s (B, Col, Hi, Pre, Cap, Cloze) parameters.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): HPAA generates human-perceptible adversarial examples by combining benign and toxic text inputs. This claim is supported by experiment (E1), which reproduces the adversarial-example generation pipeline described in Section 4 of the paper.

(C2): HPAA achieves strong evasion performance against both commercial moderation systems and local safety models. This claim is supported by experiment (E2), which reproduces the detector-evaluation results reported in Section 5.

(C3): HPAA preserves human recognition while reducing detector effectiveness, as demonstrated by the user-study results reported in Table 1 and Figure 8. This claim is supported by experiment (E3), which reproduces the corresponding user-study analyses.

A.4.2 Experiments

(E1): Adversarial Example Generation [10 human-minutes + 2 compute-minutes]
Preparation: Install the artifact and dependencies as described in Section A.3.1.
Execution: Run:

```
python HPAA.py
-bc Hotel -b text
-tc Advbench_10 -t text
-m M1 -l W -s Hi
--seed 42 -p adv
```

Results: Generated adversarial examples are saved to: `./HPAA/adv.M1-W-Hi.csv`

(E2): Detector Evaluation [30 human-minutes + variable compute-hours] The runtime depends on the number of adversarial examples, the selected detector, and API response latency.
Preparation: Generate adversarial examples using Experiment (E1). Configure API credentials or local detector models.
Execution: Run:

```
python HPAA.py
-f ./HPAA/adv.M1-W-Hi.csv
-dn perspective_api
-ep verify
```

Additional detectors can be evaluated using the commands provided in `run.sh`.
Results: The evaluation results are written to: `./HPAA/verify.*.csv`

(E3): User-Study Analysis [30 human-minutes + 2 compute-hour]
Preparation: No additional preparation is required.
Execution: Follow the instructions provided in:
`user_study/SurveyDataRound1/README.md`
`user_study/SurveyDataRound2/README.md`
Results: The analysis reproduces user-study statistics.

A.5 Notes on Reusability

The artifact supports extension to new datasets, typographic configurations, and moderation systems. New datasets can be incorporated by placing the corresponding files in the appropriate data directory and registering them in `./src/gen_HPAA.py`. Additional typographic configurations can be supported by extending the generation logic in `./src/gen_HPAA.py`. New moderation systems can be integrated by implementing detector wrappers in `./src/detectors.py` and `./src/eval_HPAA.py`, and registering the detector configuration in `./src/detectors.yaml`.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: A Formal Security Analysis of CAN XL

Zhaozhou Tang
Georgia Institute of Technology

Khaled Serag
Qatar Computing Research Institute

Z. Berkay Celik
Purdue University

Vijay Ganesh
Georgia Institute of Technology

Saman Zonouz
Georgia Institute of Technology

Raheem Beyah
Georgia Institute of Technology

A Artifact Appendix

A.1 Abstract

We conduct a formal security analysis of CAN XL, the latest generation of the Controller Area Network (CAN) protocol. We model the protocol using the nuXmv model checker and discover several vulnerabilities in the standard. To demonstrate their exploitability, we evaluate several attacks comprising the discovered vulnerabilities on a testbed. In this artifact, we provide nuXmv models of CAN XL, scripts to run the analysis, and implementations of the evaluated attacks. Due to licensing restrictions of the proprietary software required for the attack experiments (explained in A.2.4), the reproducibility evaluation workflow in this document focuses on reproducing the formal analysis results and validating the proposed standard revisions.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our artifact does not perform destructive steps. Executing them does not pose security or privacy concerns to the user.

A.2.2 How to access

The artifact can be downloaded from `https://zenodo.org/records/20262062`.

A.2.3 Hardware dependencies

CAN XL Formal Analysis. The analysis can be performed on a standard PC. We used a PC with 32 GB of RAM. The compute times in A.4.2 are based on results using this PC.
Attack Experiments. Replicating them requires a Vector VN1641 CAN interface, an STM32 NUCLEO-H755ZI-Q development board, and a CAN transceiver (we used SN65HVD230). Due to difficulties explained in A.2.4, we are unable to provide access to the experiment platform.

A.2.4 Software dependencies

CAN XL Formal Analysis. We performed the analysis on a Windows 11 PC using Python 3.10.11 and nuXmv 2.1.0. The analysis can also be reproduced on a macOS or Linux machine, both of which are supported by nuXmv. On a macOS machine, libgmp needs to be installed to use nuXmv.
Attack Experiments. Running them requires a Windows PC with Vector CANoe, VN1641 driver, and STM32CubeIDE. We used CANoe 18.4.26 (SP4), VN1641 driver 24.10.10, and STM32CubeIDE 1.12.1. CANoe requires a paid and machine-bound license. Our license allows us to use CANoe only on one author's personal PC. Therefore, it is difficult to provide remote access to the experiment platform.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

After downloading the artifact archive from Zenodo, extract its contents. Install Python 3.10 or higher by following the instructions at `https://www.python.org/downloads/`. No additional Python packages are needed. Install nuXmv 2.1.0 by going to `https://nuxmv.fbk.eu/previous-version.html` and clicking on the download link for the appropriate OS. Extract the downloaded file. The nuXmv binary is located in the `bin` directory within the extracted folder. For a macOS machine, install libgmp with Homebrew using the command `brew install gmp`.

A.3.2 Basic Test

Navigate to any one of the following three directories: `models/error_signalling_enabled`, `models/error_signalling_disabled`, or `models/error_signalling_disabled_revision`, and run:

```
python run.py --nuxmv-dir
    "absolute_path_to_nuXmv_binary" --test
```

This analyzes the model of the arbitration phase and finds one possible ending state of the protocol when it starts executing from the idle state. If the setup is correct, a message similar to the following should be displayed:

```
Executing arbitration_phase.smv from
    (n1_state=idle, n2_state=idle) can reach
    (n1_state=bus_reintegration,
    n2_state=bus_reintegration).
```

A.4 Evaluation workflow

The workflow below evaluates the reproducibility of the artifact's formal analysis component, including vulnerability discovery and standard revisions, which covers the paper's most important contributions. The attack experiments are not included due to difficulties in providing access to the experiment platform, as explained in A.2.4. Nevertheless, the artifact repository includes the attack implementation and documentation for interested readers.

A.4.1 Major Claims

(C1): We formally model the CAN XL protocol and verify the properties described in Sec. 4.1 of the paper. We discover that CAN XL inherits all CAN CC vulnerabilities and introduces seven new standard-level vulnerabilities. The vulnerabilities are summarized in Tbl. 2 and described in detail in Sec. 5 of the paper. These results can be reproduced with experiments E1 and E2.

(C2): We propose standard revisions to CAN XL's error signalling disabled configuration and successfully verify $\mathcal{P}_2$ and $\mathcal{P}_3$ for the revised protocol. Therefore, the standard revisions fix *V2* without introducing new problems. The revisions are detailed in Sec. 8 of the paper and can be validated with experiment E3.

A.4.2 Experiments

(E1): [Analyzing error signalling disabled configuration] [5 human-minutes + 5 compute-minutes]:
How to: Navigate to the `models/error_signalling_disabled` directory. It contains the nuXmv models of CAN XL's error signalling disabled configuration, a `run.py` script to run the analysis, and counterexample traces we generated for each vulnerability (*V1*, *V2*, *V5*) in the *traces* directory. Run the command:

```
python run.py --nuxmv-dir
    "absolute_path_to_nuXmv_binary"
```

This command first computes reachable states of the protocol, then verifies the relevant properties ($\mathcal{P}_1$-$\mathcal{P}_6$), and generates counterexamples for each error signalling disabled vulnerability (*V1*, *V2*, *V5*).
Results: The script displays that verification for all properties fails except for $\mathcal{P}_2$. A new directory *counterexamples* is created. For each provided counterexample in the `trace` directory, there is a file with the same name and similar content in the *counterexamples* directory.

(E2): [Analyzing error signalling enabled configuration] [5 human-minutes + 20 compute-minutes]:
How to: Navigate to the `models/error_signalling_enabled` directory. It contains the nuXmv models of CAN XL's error signalling enabled configuration, a `run.py` script to run the analysis, and counterexample traces we generated for each vulnerability (*V4-V7*, *Va-Vc*) in the *traces* directory. Run the command:

```
python run.py --nuxmv-dir
    "absolute_path_to_nuXmv_binary"
```

This command first computes reachable states of the protocol, then verifies the relevant properties ($\mathcal{P}_1$-$\mathcal{P}_9$), and generates counterexamples for each error signalling enabled vulnerability (*V4-V7*, *Va-Vc*).
Results: The script displays that verification for all properties fails except for $\mathcal{P}_7$. A new directory *counterexamples* is created. For each provided counterexample in the `trace` directory, there is a file with the same name and similar content in the *counterexamples* directory.

(E3): [Analyzing the standard revisions] [5 human-minutes + 5 compute-minutes]:
How to: Navigate to the `models/error_signalling_disabled_revision` directory. It contains the nuXmv models of CAN XL's error signalling disabled configuration with our proposed standard revisions and a `run.py` script to run the analysis. It does not have a *traces* directory as the verification is successful and no counterexamples are generated. Run the command:

```
python run.py --nuxmv-dir
    "absolute_path_to_nuXmv_binary"
```

This command first computes reachable states of the protocol, and then verifies $\mathcal{P}_2$ and $\mathcal{P}_3$.
Results: The script displays that verification of both $\mathcal{P}_2$ and $\mathcal{P}_3$ succeeds. No counterexamples are generated and no *counterexamples* directory is created.

A.5 Notes on Reusability

Our artifact can be reused to analyze other aspects of CAN XL. New properties can be added to the nuXmv models in the `template` directory for each error signalling configuration. The property names need to be added to the `properties` field of the `model` variable at the beginning of `run.py`. The analysis can then be performed by invoking `run.py` using the same commands documented in A.4.2.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: SoK: The Pitfalls of Deep Reinforcement Learning for Cybersecurity

Shae McFadden[†‡§], Myles Foley[‡*], Elizabeth Bates[‡], Ilias Tsingenopoulos[¶], Sanyam Vyas[‖‡], Vasilios Mavroudis[†], Chris Hicks[‡], Fabio Pierazzi[§]
[†]*King's College London*, [‡]*The Alan Turing Institute*, [§]*University College London*, [‖]*Cardiff University*, [¶]*KU Leuven*, [*]*Devotion AI Labs*

A Artifact Appendix

A.1 Abstract

Our artifact provides the code required to run and reproduce the results of the four case studies in our paper. Each case study corresponds to a specific subset of the 11 pitfalls identified across the four stages of agent development (modeling, training, evaluation, deployment):

AutoRobust (adversarial malware creation): This case study demonstrates the Gain Attribution (E-GA) pitfall by disentangling the contribution of the state/action space from the learned policy, by comparing MDP, POMDP, and Random policies.

Link (web security testing, XSS): This case study demonstrates the Modeling Correctness (M-MC) and Partial Observability (M-PO) pitfalls by comparing the original MDP formulation against a corrected Distinct States formulation.

SQIRL (web security testing, SQL injection): This case study demonstrates the Environment Complexity (E-EC) pitfall by evaluating agents on simplified (non-sanitized) versus realistic (sanitized) endpoints.

MiniCAGE (autonomous cyber defense): This case study demonstrates the Hyperparameter Reporting (T-HR), Variance Analysis (T-VA), Policy Convergence (T-PC) training pitfalls and the Underlying Assumptions (D-UA), Non-Stationarity (D-NS) deployment pitfalls.

Each case study directory contains the modified environment, the training/evaluation scripts, the MDP specification, and a `case_study.diff` documenting all changes from the original open-source repository released with their respective papers. The *reproducibility* directory provides scripts that build per-study Python environments and rerun both quick (single-seed) and full (20-seed) results.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact includes deliberately vulnerable web-application targets (WAVSEP for Link and the SMB for SQIRL) and an environment that learns to evade a ML-based malware detector (AutoRobust). These components are intended only for local, isolated evaluation:

- The WAVSEP and SMB containers should **not** be exposed on externally reachable ports, as they contain intentionally vulnerable code. They are bound to `localhost` by default.
- The AutoRobust case study trains agents to modify dynamic-analysis reports to evade detection. No live malware is executed; the artifact operates on pre-collected dynamic-analysis report features. The malware/goodware dataset needs to be downloaded from Kaggle.

Overall, no destructive operations are performed on the host, and all exploitation of vulnerabilities is contained within docker containers.

A.2.2 How to access

The final artifact will be made available on Github and Zenodo, updating the current availability artifacts:

- https://doi.org/10.5281/zenodo.20209122
- https://github.com/alan-turing-institute/DRL4Sec-Pitfalls

A.2.3 Hardware dependencies

No special hardware is required, however a CUDA-compatible GPU is *recommended* for the AutoRobust case study. The full reproduction of 20 seeds is computationally-intensive, therefore, we recommend running a single seed (`reproducibility/reproduce-one-seed.sh`).

A.2.4 Software dependencies

- `python3.9` & `python3.10` are both required as the case studies have varying dependency requirements
- `Docker Engine` & `Docker Compose` are required for the Link and SQIRL case studies.
- `Node.js` & `Chrome`/`Chromium` with its system libraries are required for the Link case study.

- The python dependencies for each of the case studies can be found in `reproducibility/envs/{case-study-name}.txt`, with the venv setup automated by `reproducibility/setup-envs.sh`.

A.2.5 Benchmarks

- *AutoRobust:* the dataset needs to be downloaded from Kaggle and placed in `AutoRobust-Case-Study/data/`. This is used for fine-tuning the malware classifier and the agent's attacks. The dataset can be found at: https://www.kaggle.com/datasets/greimas/malware-and-goodware-dynamic-analysis-reports
- *Link:* no external data required, the WAVSEP docker file is provided.
- SQIRL*:* no external data required, SQLi MicroBenchmark (SMB) docker file is provided.
- *MiniCAGE:* the MiniCAGE ACD environment is provided. The `rl-reliability-metrics` package is cloned and installed automatically by `setup-envs.sh`.

A.3 Set-up

A.3.1 Installation

Clone the repository:

- `git clone https://github.com/alan-turing-institute/DRL4Sec-Pitfalls`
- `cd DRL4Sec-Pitfalls`

Then to create the case study virtual environments:

- `cd reproducibility`
- `chmod +x *.sh`
- `./setup-envs.sh`

This creates a venv for each case study and the reproduction scripts automatically activate the correct venv for each experiment.

A.3.2 Basic Test

After `setup-envs.sh` has created the four venvs (and Docker is running), the basic functionality test can be run as follows:

- `./smoke-test.sh`, or
- `./smoke-test.sh {case-study-name}`.

This script checks that every case study is functional by: checking the imports, docker containers, and running a truncated version of the experiments to ensure the full pipelines are working. If called with the name of a specific case study (`autorobust`, `link`, `sqirl`, or `minicage`) then the script will only check that case study.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Gain Attribution (E-GA), AutoRobust.* The observed performance of an agent can be misattributed to policy learning when environment design may provide a large fraction of that performance. Through evaluating the MDP, POMDP, and Random policies, comparisons between random and the other policies demonstrates that the design of the environment provides substantial evasion success even without a learned policy.

(C2): *Modeling Correctness & Partial Observability (M-MC, M-PO), Link.* Correcting a conflated state-space formulation improves performance. The corrected *Distinct States* formulation increases the percentage of vulnerabilities found over the Original formulation by 10.1% and raises the lower bound of the 95% confidence interval by 17.3% across 20 runs.

(C3): *Environment Complexity (E-EC),* SQIRL. Performance measured in a simplified environment is over-inflated relative to a realistic one. Agents that never experience sanitized (defended) endpoints during training suffer up to a 47.4% drop in vulnerabilities found when evaluated against sanitized defenses.

(C4): *Training & Deployment pitfalls (T-HR, T-VA, T-PC / D-UA, D-NS), MiniCAGE.* (i) A single hyperparameter change can drastically alter performance, and evaluating over many runs with confidence intervals reveals a wide behavioral distribution. (ii) Minor violations of deployment assumptions (changing red/blue agent action ordering) and non-stationary adversaries (evaluating against unseen attacker strategies) severely degrade deployed-agent performance.

A.4.2 Experiments

The reproduction of all four case studies is automated through the same reproduction scripts, therefore, after initial setup only the following scripts need to be run:

- `reproducibility/reproduce-one-seed.sh`: reproduces all settings/experiments in the paper with one seed/agent each; pass case study name to run only that case study. *Parallel*: run the script in parallel terminals once for each case study (1 day). *Sequential*: run the script once without specifying case study (2-3 days).
- `reproducibility/reproduce-full.sh`: full paper scale reproduction with 20 seeds/agents per configuration (≈weeks).
- `reproducibility/results-summary.md`: contains the output from the reproducibility scripts. If they were run in parallel, you may need to activate the `minicage venv` and run `python extract-results.py` to generate the full report.

Each script activates the correct venv per case study, starts/tears down the Docker containers, runs training and evaluation, and writes a results summary. The per-study commands remain available for running an individual study in isolation, but are not required.

(Setup): [≈ 20 human-minutes + ≈ 1 compute-hour] Complete A A.3.1 and A A.3.2.

(Run) *[≈ 5 human-minutes + ≈ 1 parallel-compute day or ≈ 3 compute days]* run one seed reproducibility script described above and then the results will be outputted into `reproducibility/results-summary.md`.

(E1): *AutoRobust [≈ 1 compute-day]* See the AutoRobust table in `results-summary.md` (corresponding to Figure 3 in the paper). The mean malware score of the *Random* policy relative to the trained *MDP/POMDP* policies demonstrate that a large portion of the performance is attributable to the state/action space design rather than learning.

(E2): *Link [≈ 8 compute-hours]* See the Link table in `results-summary.md` (corresponding to Table 1 in the paper). the *Distinct States* formulation should improve vulnerabilities found and raise the 95% CI lower bound over the *Original* formulation.

(E3): SQIRL*[≈ 8 compute-hours]* See the SQiRL table in `results-summary.md` (corresponding to Table 2 in the paper). Comparing the *Eval Sanitized* and *Eval Non-Sanitized* columns, agents trained only on non-sanitized endpoints should show a larger drop in vulnerabilities found against sanitized endpoints.

(E4): *MiniCAGE [≈ 6 compute-hours]* See the MiniCAGE tables tables in `results-summary.md` (corresponding to Figure 2 & Table 3 in the paper). The hyperparameter ablation table reports the performance at the end of training, corresponding to the right most point of Figure 2 in the paper. This table demonstrates that changing even a single hyperparameter can drastically change performance and (if run on multiple seeds) demonstrates across the seeds to motivate variance analysis. The next two tables in `results-summary.md` corresponding to the top and bottom sections of Table 3 in the paper. These demonstrate the degradation of performance in the off-diagonal results which correspond to when underlying assumptions breakdown or non-stationarity is unhandled.

A.5 Notes on Reusability

Each case study is self-contained and can be run independently of the others. The `Modified-*/` directories provide the stable version of the environments used in our evaluations with `case_study.diff` isolating the exact modifications made to the original code bases. The `reproduce-*.sh` scripts can be scaled down for quick experimentation (fewer seeds/timesteps) or up to the paper configuration.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: InstrSem: Automatically and Generically Inferring Semantics of (Undocumented) CPU Instructions

Lorenz Hetterich, Fabian Thomas, Tristan Hornetz, Michael Schwarz
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This artifact provides the source code of InstrSem. InstrSem is a tool that allows automatic reverse-engineering of undocumented instructions across a range of architectures. Alongside the source code of InstrSem, we provide the required scripts to reproduce the main claims of the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact contains proof-of-concepts to show how InstrSem can reverse-engineer a wide range of instructions. The experiments mostly run in QEMU inside a Docker container and should not affect the host device. No personal data is collected or transmitted in any way.

A.2.2 How to access

The artifact is available on Zenodo:
https://zenodo.org/records/17974657

A.2.3 Hardware dependencies

The artifact was tested on an x86-64 based machine running Ubuntu 24.04. The experiments required to verify the main claims of the paper do not rely on specific hardware. For completeness, the code for experiments relying on specific hardware is also included in the artifact.

A.2.4 Software dependencies

The following software is required on the reviewer's machine.

- `python3`.
- `docker`: The active user must be part of the `docker` group.
- `sh`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Download the artifact from Zenodo and extract it. Navigate to the root of the artifact and run the `setup_docker.sh` script. This script sets up the required docker container. Most experiments can then be run by executing a python script.

A.3.2 Basic Test

After setting up the docker container, a basic test is provided that reverse-engineers multiple instructions across different architectures.

Execution:

```
python3 basic_demonstration.py
```

Execution may take up to two hours.

Expected results: The output should contain semantics for the reverse-engineered instructions. It should roughly match the sample output provided in `results/basic_demonstration.txt`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): InstrSem is generic and can recover and generalize instruction semantics across ISAs (**E1, E2, E3, E4**).
(C2): InstrSem can uncover the semantics of undocumented LoongArch64 vector instructions (**E4**).
(C3): InstrSem can recover 34 of 38 64-bit RISC-V integer instructions (**E2**).
(C4): InstrSem is also applicable to complex ISAs like x86-64 (**E3**).
(C5): InstrSem can be run on a full RISC instruction set (**E4**).

(C6): The amount of covered encodings drastically increases due to InstrSem's generalization step (**E4**).
(C7): InstrSem triggers QEMU crashes (**E5**).

A.4.2 Experiments

If required, more information for each experiment is provided in the `README.md` file of the artifact. Additional experiments that rely on special hardware and are not required to verify the main claims of the paper are also described in the `README.md` file of the artifact.

(E1): [basic_demonstration] [5 human-minutes 2 compute-hours]: Basic demonstration of InstrSem.
Preparation:

```
sh setup_docker.sh
```

Execution:

```
python3 basic_demonstration.py
```

Results: Instruction semantics for multiple instructions should be successfully recovered and generalized. The output should roughly match the sample output provided in `results/basic_demonstration.txt`.

(E2): [eval_riscv64] [5 human-minutes 2 compute-hours]: Recovering the semantics of RISC-V 64-bit Integer Instructions.
Execution:

```
python3 eval_riscv64.py > table_rv64.md
```

Results: The resulting table in `table_rv64.md` should match the sample table provided in `results/eval_riscv64.md` apart from possible differences in execution time. In rare cases, `slti` or `sltiu` can fail to recover and generalize semantics successfully. If this is the case, you can repeat the experiment.

(E3): [eval_x86-64] [5 human-minutes 2 compute-hours]: Recovering the semantics of common x86-64 instructions.
Execution:

```
python3 -u eval_x86-64.py > table_x86-64.md
```

Results: The resulting table in `table_x86-64.md` should match the sample table provided in `results/eval_x86-64.md` apart from possible differences in execution time.

(E4): [eval_loongarch64] [5 human-minutes 10 compute-hours]: Running InstrSem on the full LoongArch instruction space.
Execution:

```
python3 eval_loongarch64.py
```

We encourage you to run this evaluation for 10h but shorter or longer runs are also possible. When you are done, you can use CTRL+C to terminate the evaluation. This might still lead to an active docker container which you can terminate. You can also interrupt the evaluation using CTRL+C (only shortly press it to raise a single `KeyboardInterrupt`). Then, when you later start the script again, it should continue instead of starting from scratch.
Results: The output of InstrSem for newly discovered instructions is written to one file in the `out` directory for each instruction. We expect a large number of files to exist in this directory. Further, whenever a new instruction is discovered, the script prints the runtime (in seconds) and total number of recovered encodings to stdout. If you run the evaluation for 10 hours, we expect at least 600 million instruction encodings to be covered.
Further, we expect the semantics of several vector instructions to be recovered. To filter for possible vector instructions, you can use the following commands:

```
grep -r out -e vector
```

(E5): [eval_qemu_crash] [1 human-minute]: QEMU crashes.
Execution:

```
python3 eval_qemu_crash.py
```

Results: The execution should print multiple error-messages:

```
ERROR:../target/loongarch/tcg/insn_trans/
    trans_vec.c.inc:3574:vldi_get_value: code
    should not be reached
Bail out! ERROR:../target/loongarch/tcg/
    insn_trans/trans_vec.c.inc:3574:
    vldi_get_value: code should not be reached
**
```

Finally, InstrSem will give up recovering semantics with a `too many sample collections failed!` exception:

```
Exception: too many sample collections 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: HAMLOCK: HArdware-Model LOgically Combined attacK

Sanskar Amgain[1*], Daniel Lobo[2*], Atri Chatterjee[2*], Swarup Bhunia[2], Fnu Suya[1]

[1]University of Tennessee [2]University of Florida

A Artifact Appendix

A.1 Abstract

This artifact implements HAMLOCK, a hardware–software co-design backdoor in which a small software-side trigger pathway is paired with a hardware Trojan payload, so that the backdoored model behaves benignly in software-only inspection yet misclassifies triggered inputs once deployed on the Trojaned hardware. It provides the three attack variants studied in the paper—single-neuron trigger optimization, single-neuron weight optimization, and the multi-neuron attack—together with a functional emulation of the hardware Trojan (trigger detection via sign-/exponent-bit comparison and payload bias injection), so no physical FPGA or ASIC is required to reproduce the model-level results. The artifact further includes evaluation harnesses for six representative defenses spanning black-box and white-box backdoor-sample detection (STRIP, BBCaL, IBD-PSC, TED) and lightweight backdoor mitigation (fine-tuning, fine-pruning, and CLP). Running the artifact trains the clean models, injects the backdoors, and produces the backdoored checkpoints and per-defense logs that reproduce the paper's major claims (C1–C4) on MNIST, CIFAR-10, GTSRB, and ImageNet with LeNet, VGG-16, and ResNet-18. Reproducing claims C1–C4 requires only a single CUDA-capable GPU and the open-source software in `requirements.txt`.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Executing this artifact poses no risk to the evaluator's machine. The artifact performs no destructive operations, requires no elevated privileges, and does not disable, weaken, or interfere with any security mechanism on the host system. It does not establish network connections beyond downloading the standard public benchmark datasets and pretrained models, and it neither collects nor transmits any user or system data. All computation is confined to the project directory and standard framework caches.

All experiments use only publicly available, non-sensitive benchmark datasets (MNIST, CIFAR-10, GTSRB, and ImageNet), and no personally identifying or private information is involved at any stage.

The artifact implements a backdoor attack and therefore has inherent dual-use characteristics. However, it produces only backdoored model checkpoints and an emulated hardware-Trojan inference pipeline within an isolated research environment; it does not generate malware, exploit any external system, or target deployed hardware. Consistent with the Ethical Considerations in our paper, the artifact is released solely to enable defensive research into the hardware–software attack surface, and we explicitly discourage any malicious use.

A.2.2 How to access

The artifact is available at https://doi.org/10.5281/zenodo.17980434. The repository contains the attack implementations and entry points, the six defense evaluation harnesses under `defenses/` (each with its own `README.md`), the RTL under `rtl/`. Datasets and the ImageNet-pretrained backbones are downloaded automatically as described in the Benchmarks section (ImageNet itself must be obtained separately). We will provide stable updated reference for the camera-ready version once all the reviewer's feedback has been addressed.

A.2.3 Hardware dependencies

Reproducing our major claims (C1–C4) does *not* require any specialized hardware accelerator. The hardware Trojan logic (trigger detection via sign-/exponent-bit comparison and payload bias injection into the output logits) is functionally emulated within the software inference pipeline, so no physical FPGA or ASIC is needed to obtain the reported attack-success and clean-accuracy results.

The model-level experiments require a single machine with one CUDA-capable NVIDIA GPU. The MNIST (LeNet), CIFAR-10, and GTSRB (VGG-16/ResNet-18) experiments are lightweight and run comfortably on a GPU with at least 8 GB of memory; a CPU-only run is possible for these but considerably slower. The ImageNet experiments are the most

*These authors contributed equally to this work.

demanding and we recommend a GPU with at least 16–24 GB of memory for these.

Our experiments were conducted on a server equipped with four NVIDIA L40S GPUs (48 GB each; only one is required to reproduce our results), 128 logical CPU cores, and approximately 512 GB of system RAM. No specific microarchitecture, interconnect, or hardware performance counters are required.

A.2.4 Software dependencies

The artifact runs on a Linux system (tested on Ubuntu) and requires *Python 3.9.25*. All Python dependencies, including PyTorch with CUDA support, are listed in *requirements.txt* and can be installed into an isolated virtual environment as described in the Installation section. A working NVIDIA CUDA driver compatible with the installed PyTorch build is required for GPU execution. Reproducing the model-level experiments (claims C1–C4) requires only the above open-source software.

A.2.5 Benchmarks

We evaluate on four standard, publicly available image-classification benchmarks: MNIST, CIFAR-10, GTSRB, and ImageNet (ILSVRC-2012). MNIST, CIFAR-10, and GTSRB are downloaded automatically into the directory specified by `-dataset_dir` on first use (the directory is created if it does not exist). ImageNet must be obtained manually from its official source (https://www.image-net.org/challenges/LSVRC/2012/2012-downloads.php) and placed in the expected directory, as it requires registration and is not auto-downloaded.

No backdoored or task-specific model checkpoints need to be supplied. The convolutional backbones (VGG-16 and ResNet-18) are initialized from torchvision's standard ImageNet-pretrained weights, which are downloaded automatically on first use; LeNet (for MNIST) is trained from scratch. The artifact then trains/fine-tunes the clean models locally during the trigger-optimization stage, and the resulting checkpoints are reused by the weight-optimization and multi-neuron attacks.

A.3 Set-up

A.3.1 Installation

Our experiments were conducted using *Python 3.9.25*. We recommend creating an isolated virtual environment before installing the dependencies. The required packages are listed in *requirements.txt* and can be installed with the following commands:

```
python3 -m venv .env
source .env/bin/activate
python3 -m pip install -r requirements.txt
```

A.3.2 Basic Test

The following one-epoch run exercises the full software path—dataset download, model construction, GPU training, backdoor injection, the emulated trigger pathway, and checkpoint serialization—in a few minutes on a single GPU. It is a scaled-down (`-epochs 1`) invocation of the trigger-optimization attack and writes its checkpoints to a throwaway directory.

```
python3 main.py \
    --dataset_dir ./data/ --dataset cifar10 \
    --epochs 1 --model resnet --device cuda:0 \
    --target_label 0 --inject 1 \
    --train_model 1 --batch_size 256 \
    --model_path ./checkpoints_smoketest \
    --dump_model 1 --lam 0.1 --threshold 0.0 \
    --use_normalization 1 --seed 1
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): HAMLOCK achieves a near-100% attack success rate (ASR) while incurring only a minimal clean-accuracy drop (at most $\sim$2.6%). This is demonstrated by experiment (E1) described in Section 5.1, with results reported in Table 1.

(C2): HAMLOCK evades state-of-the-art white-box backdoor-sample detection defenses (IBD-PSC and TED). This is demonstrated by experiment (E2) described in Section 5.2 (subsection *Backdoor sample detection*), with results reported in Table 3.

(C3): HAMLOCK evades black-box backdoor-sample detection defenses (STRIP and BBCAL), both before deployment (software-only model) and after deployment on the Trojaned hardware. This is demonstrated by experiment (E3) described in Section 5.3, with results reported in Table 4 and Table 5.

(C4): HAMLOCK remains effective against backdoor-mitigation defenses based on lightweight retraining (fine-tuning and fine-pruning). This is demonstrated by experiment (E4) described in Section 5.2 (subsection *Effectiveness under lightweight retraining*), with results reported in Table 6.

A.4.2 Experiments

(E1): *Attack effectiveness without defenses [30 human-minutes + 4–6 compute-hours (CIFAR-10 and GTSRB) + $\sim$10 GB disk]*: This experiment supports claim C1. It runs the three attack variants of HAMLOCK—the single-neuron trigger-optimization attack (`1N-trigger`), the single-neuron weight-optimization attack (`1N-weight`), and the three-neuron multi-neuron attack (`3N-weight`)—on a given dataset/architecture and reports, for the resulting backdoored model, the clean accuracy and the attack

success rate (ASR) both *without* the Trojaned hardware (software-only model) and *with* the emulated hardware Trojan active. The expected outcome is that the software-only model is functionally indistinguishable from a clean model (clean-accuracy drop of at most ∼2.6% and an ASR of effectively 0, i.e. at most ∼0.6%), while the same model deployed on the emulated Trojaned hardware reaches an ASR of ∼100% (slightly lower for the `3N-weight` variant). These values populate Table 1.

How to: The three attacks are run, *in this order*, using the prebuilt scripts in `scripts/` and documented in the repository `README.md` (section "Steps to run the attack"): `run_trigger_optimization_attack.sh` (`1N-trigger`), `run_weight_optimization_attack.sh` (`1N-weight`), and `run_3N_attack.sh` (`3N-weight`). The order is mandatory: the trigger-optimization step trains and saves the clean checkpoint (under `checkpoints/clean_models_1/`) that the other two attacks load. Select the (dataset, architecture) cell by editing the `dataset` and `model` variables (and `-device`) at the top of each script; the full argument list is described in the `README.md` "Arguments" table.

Preparation: Activate the virtual environment from the Set-up section. Datasets are handled as described under Benchmarks (MNIST, CIFAR-10, GTSRB auto-download; ImageNet placed manually). No pretrained checkpoints are required—they are produced by the first script. We recommend reproducing the CIFAR-10 and GTSRB rows (lightweight, single GPU).

Execution: From the repository root:

```
bash ./scripts/run_trigger_optimization_attack.sh
bash ./scripts/run_weight_optimization_attack.sh
bash ./scripts/run_3N_attack.sh
```

Repeat (after editing `dataset`/`model`) for each Table 1 row to reproduce.

Results: Each script prints the two Table 1 metrics to standard output. The mapping is the only non-obvious part:

- **Clean accuracy (CA), w/o hardware:** `Accuracy AFTER injection` (trigger/weight scripts) or `Clean acc after` (3N). The clean-accuracy *drop* is relative to `Accuracy BEFORE injection` (for 3N, relative to the clean checkpoint's accuracy from the first script).
- **ASR with hardware Trojan active (∼100%):** `ASR` (trigger script) or `ASR on triggered images` (weight/3N) — the fraction of *triggered* inputs misclassified to the target label once the emulated payload fires (the monitored neuron exceeds the trigger threshold). This is the ASR reported in Table 1.
- **Clean-input false-trigger rate (software-only, ≤0.6%):** printed as `ASR on clean images` (weight/3N) — despite that label it is *not* an ASR but a false-positive rate: the fraction of *clean* inputs on which the monitored neuron spuriously exceeds the threshold. A near-zero value confirms the dormant software-only model does not fire the backdoor on benign inputs.

Expect, for every cell, with-hardware ASR ≈100% (somewhat lower for `3N-weight`), without-hardware ASR near 0, and a small clean-accuracy drop (within the per-dataset bounds above). This validates claim C1.

(E2): *Evading white-box backdoor-sample detection (IBD-PSC and TED) [30 human-minutes + 2–4 compute-hours + negligible extra disk (reuses E1 checkpoints)]*: This experiment supports claim C2. It runs the two white-box, input-level detectors—IBD-PSC and TED—against the backdoored *software-only* models from E1 (these detectors require access to model internals, so they are applied to the dormant pre-deployment model). The expected outcome is that neither detector can separate backdoored from clean inputs, yielding an AUROC of ≈0.5 (no better than random). These results populate Table 3.

How to: Both defenses reuse the checkpoints produced in E1 (no retraining). The detectors and their exact commands are documented in `defenses/IBD-PSC/README.md` and `defenses/TED/README.md`, and are driven by the prebuilt sweep scripts in each directory. Each run prints an AUROC that is compared against Table 3.

Preparation: Activate the environment from the Set-up section and ensure E1 has been run so the backdoored checkpoints exist under `checkpoints/` (see the per-defense `README.md` for the exact path layout). IBD-PSC is evaluated on the single-neuron attacks (`hamock`, `hamock_weights`); TED additionally covers the multi-neuron attack (`hamock_sep`). We recommend the CIFAR-10 and GTSRB configurations (ResNet-18 / VGG-16).

Execution: From the repository root:

```
bash defenses/IBD-PSC/run.sh
bash defenses/TED/run.sh
```

Each script writes one log per (attack, dataset, model) into the respective `logs/` directory.

Results: Read the AUROC from each run's output (IBD-PSC prints a summary line plus explicit `AUROC:`/`TPR:`/`FPR:` lines; TED prints `AUC:` with TP/FP/TN/FN). For every configuration the AUROC should be ≈0.5 (observed range ∼0.43–0.52), with low TPR. TED's `Accuracy on bd_loader` is also low, confirming the software model does not act on the trigger. This shows HAMLOCK's backdoored inputs are indistinguishable from clean inputs to both white-box detectors, validating claim C2.

(E3): *Evading black-box backdoor-sample detection (STRIP and BBCaL) [30 human-minutes + 2–4 compute-hours + negligible extra disk (reuses E1 checkpoints)]*: This experiment supports claim C3. It runs the two black-box,

input-level detectors—STRIP and BBCaL—against the backdoored models from E1, in both the *software-only* scenario (the dormant model, before hardware deployment) and the *hardware-deployed* scenario (with the Trojan functionally emulated in the inference pipeline). The expected outcome is that both detectors fail to separate backdoored from clean inputs in both scenarios, yielding an AUROC of ≈0.5 (no better than random). These results populate Table 4 (MNIST) and Table 5 (GTSRB and CIFAR-10).

How to: Both defenses reuse the checkpoints produced in E1 (no retraining). The detectors and their exact commands are documented in `defenses/strip/README.md` and `defenses/bbcal/README.md`, and are driven by the prebuilt sweep scripts in each directory. Each run prints a summary line whose AUROC field is the metric compared against Table 4/5.

Preparation: Activate the environment from the Set-up section and ensure E1 has been run so the backdoored checkpoints for `<attack>` ∈ {`hamock`, `hamock_weights`, `hamock_sep`} exist under `checkpoints/` (see the per-defense `README.md`). As in the READMEs, we recommend the CIFAR-10 and GTSRB configurations (ResNet-18 / VGG-16).

Execution: From the repository root, run each defense's script; each one covers all scenarios (software-only, hardware single-neuron, and hardware multi-neuron):

```
bash defenses/strip/run.sh
bash defenses/bbcal/run.sh
```

Each script writes one log per (attack, dataset, model) into the respective `logs/` directory.

Results: Read the AUROC from each run's final summary line (`tn fp fn tp f1 precision recall AUROC clean_acc` for STRIP; `... AUROC` plus explicit `AUROC:`/`TPR:`/`FPR:` lines for BBCaL). For every configuration the AUROC should be ≈ 0.55 (observed range ∼0.45–0.65), with recall/TPR either low or—for BBCaL—offset by an equally high FPR. This confirms that HAMLOCK's backdoored inputs are statistically indistinguishable from clean inputs to both detectors, before and after hardware deployment, validating claim C3.

(E4): *Robustness to lightweight backdoor mitigation (fine-tuning, fine-pruning, and CLP) [45 human-minutes + 4–8 compute-hours + negligible extra disk (reuses E1 checkpoints)]*: This experiment supports claim C4. It applies three model-hardening mitigations to the backdoored models from E1—fine-tuning (FT), fine-pruning (FP), and Channel Lipschitzness-based Pruning (CLP)—and measures the attack success rate after mitigation. The expected outcome is that HAMLOCK *survives* all three: the trigger neuron and the hardware payload are decoupled from the normal forward pass, so retraining/pruning on clean data leaves the trigger pathway intact. The ASR therefore stays high (≈100% for the single-neuron attacks, ≈90–98% for the multi-neuron attack) while clean accuracy is maintained. These results populate Table 6.

How to: All three mitigations reuse the E1 checkpoints (no attack re-run). The commands are documented in `defenses/finetuning_finepruning/README.md` (FT and FP) and `defenses/CLP/README.md` (CLP), and are driven by the prebuilt scripts in each directory. Each run prints the post-mitigation clean accuracy and ASR, transcribed into Table 6.

Preparation: Activate the environment from the Set-up section and ensure E1 has produced the backdoored checkpoints for `<attack>` ∈ {`hamock`, `hamock_weights`, `hamock_sep`} under `checkpoints/` (see the per-defense `README.md`). We recommend the CIFAR-10 and GTSRB configurations (ResNet-18 / VGG-16).

Execution: From the repository root. Fine-tuning and fine-pruning (single-neuron via `expt_defense.py`, multi-neuron via `expt_defense_sep.py`; select the method with `-exp finetuning` or `-exp finepruning`):

```
bash defenses/finetuning_finepruning/run.sh
```

CLP (single-neuron and multi-neuron in one script):

```
bash defenses/CLP/run.sh
```

Each script writes one log per (attack, dataset, model) into the respective `logs/` directory.

Results: For every run, read the post-mitigation clean accuracy and ASR from the log (FT/FP print `Accuracy: ..., ASR: ...`; CLP prints `Test clean accuracy` together with `Test attack success rate`, before and after pruning). The ASR should remain ≈100% for the single-neuron attacks and ≈90–98% for the multi-neuron attack, essentially unchanged from the pre-mitigation value, with clean accuracy preserved. This confirms that lightweight retraining and pruning do not remove the backdoor, validating claim C4.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: LoongLeak: Architectural Cross-Privilege-Boundary Data Leakage on LoongArch CPUs

Lorenz Hetterich, Tristan Hornetz, Fabian Thomas, Michael Schwarz
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This artifact provides proof-of-concept (PoC) implementations for LoongLeak, an architectural vulnerability affecting Loongson 3A5000 and 3A6000 CPUs. The artifact includes several case studies: a basic leakage example, evaluation for leakage speed and accuracy, a covert channel between processes, a proof-of-concept breaking ASLR and leaking stack canaries, a proof-of-concept leaking partial root password hashes, and a proof-of-concept extracting disk encryption keys. Additionally, it contains a differential fuzzer used to discover the vulnerability.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact contains proof-of-concept exploits. Running these on your own machines (if you have Loongson hardware) will demonstrate that sensitive data (like kernel memory or other users' data) can be leaked. One PoC (`sudo_hash_poc`) specifically targets the root password hash. The vulnerability was responsibly disclosed and is under embargo until the first conference day.

A.2.2 How to access

The artifact is available on Zenodo:
https://zenodo.org/records/17974800

A.2.3 Hardware dependencies

The artifact requires Loongson CPUs based on the LoongArch ISA. Specifically:

- **Loongson 3A5000**: Used for most experiments (referenced as `lab86` in instructions).
- **Loongson 3A6000**: Used for the `sudo_hash_poc` experiment (referenced as `lab85`).

Reviewers are provided with SSH access to these two machines. The private key (`artifact`) and SSH configuration (`ssh_config`) are contained in the artifact and will be valid during the artifact evaluation period only.

A.2.4 Software dependencies

The following software is required on the reviewer's machine (for cross-compilation). The instructions below assume Ubuntu 24.04:

- `python3` (standard `python3` package)
- `termcolor` Python package
- `qemu-loongarch64` (`qemu-user` package)
- `loongarch64-linux-gnu-gcc` (`gcc-14-loongarch64-linux-gnu` package). After installation, create a symlink (as root):

```
ln -s /usr/bin/loongarch64-linux-gnu-gcc-14 \
    /usr/bin/loongarch64-linux-gnu-gcc
```

- `binutils-loongarch64-linux-gnu`
- `make`, `git`, `unzip`, `zip`

The provided machines run Loongnix-20.6 (Linux kernel 4.19) and have all required software installed.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Navigate to the root of the artifact. Most components can be compiled by running `make` in their respective subdirectories. For cross-compilation, ensure the `loongarch64-linux-gnu-gcc` compiler is in your path.

A.3.2 Basic Test

The most basic test is the `leak_primitive`. It demonstrates leakage within a single process.
Execution:

```
cd leak_primitive; make
scp primitive lab86:
ssh lab86 ./primitive
```

Expected results: The output should show multiple 32-byte blocks. Some rows should contain the sequence `1000 1001 ... 101f`, which is the data from the target cache set in the victim buffer.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): LoongLeak can leak data across security domains (kernel, other userspace processes) without side channels. (Proven by E1, E3, E4, E5, E6)
(C2): The leakage is fast, achieving rates beyond 300 MB/s. (Proven by E2)
(C3): LoongLeak allows targeted leakage of specific cache lines and ways. (Proven by E1)
(C4): LoongLeak can break traditional software defenses like ASLR and stack canaries. (Proven by E4)
(C5): LoongLeak can be discovered using differential fuzzing between hardware and QEMU. (Proven by E7)

A.4.2 Experiments

More detailed instructions for each experiment are provided in the `README.md` file within the artifact.
(E1): [leak_primitive] [5 human-minutes]: Basic demonstration of LoongLeak.
Preparation:

```
cd leak_primitive; make
scp primitive lab86:
```

Execution:

```
ssh lab86 ./primitive
```

Results: Successful leakage of the target pattern `1000 1001 ... 101f`.
(E2): [evaluate_primitive] [5 human-minutes]: Measure leakage speed and accuracy.
Preparation:

```
cd evaluate_primitive; make
scp eval_primitive lab86:
```

Execution:

```
ssh lab86 ./eval_primitive
```

Results: Output shows `correct` and `time`. Leakage rate (in $\frac{MB}{s}$) is calculated as:

$$\frac{16384000000 \cdot 100000000}{time \cdot 1024 \cdot 1024}$$

Expected rate >300$\frac{MB}{s}$, accuracy ($\frac{correct}{16384000000}$) >80%.
(E3): [covert_channel] [5 human-minutes]: Data transfer between two processes.
Preparation:

```
cd covert_channel; make;
scp sender receiver lab86:
ssh lab86 "echo 'Hello World' > in.txt"
```

Execution:
In terminal 1:

```
ssh lab86 taskset -c 0 ./receiver out.txt
```

In terminal 2:

```
ssh lab86 taskset -c 0 ./sender in.txt
```

The sender in terminal 2 might not terminate automatically. Thus, you may need to terminate it manually.
Results: `out.txt` on `lab86` matches the `in.txt`.
(E4): [aslr_break] [5 human-minutes + 15 compute-minutes]: Leak stack canary and break ASLR of FFmpeg.
Preparation:

```
scp -r aslr_break lab86:
```

On lab86:

```
cd aslr_break
make
git clone --recursive \
  https://github.com/FFmpeg/FFmpeg.git \
  --branch=n7.1.1 --depth=1
cd FFmpeg
git apply ../0001-Print-Stack-Canary.patch
./configure --target-os=linux \
  --arch=loongarch \
  --prefix=/home/reviewer/ffmpeg \
  --extra-cxxflags="-fstack-protector-all -g" \
  --extra-cflags="-fstack-protector-all -g" \
  --enable-gpl --disable-stripping \
  --enable-shared --disable-static
make -j$(nproc)
make install
```

Execution: In Terminal 1:

```
ssh lab86 'cd aslr_break;
  taskset -c 0 sh start_victim.sh'
```

In Terminal 2:

```
ssh lab86 'cd aslr_break;
    taskset -c 0 ./canary_leak;
    taskset -c 0 ./aslr_break $(pidof ffmpeg);
    killall ffmpeg'
```

Results: Leaked canary matches the one printed by the victim; `aslr_break` correctly estimates the stack address.

(E5): [sudo_hash_poc] [10 human-minutes]: Leak partial root password hash on 3A6000.

Preparation:

```
cd sudo_hash_poc; make;
scp sudo_hash_poc pwd.sh lab85:
```

Execution: In Terminal 1:

```
ssh lab85 "sh pwd.sh 1; killall sh"
```

In Terminal 2:

```
ssh lab85 taskset -c 0 ./sudo_hash_poc
```

Once you see leaked password hashes in terminal 2, you can terminate `sudo_hash_poc` in terminal 2 and `pwd.sh` in terminal 1.

Results: Output contains substrings of the root password hash (`$6$7oJFvh5yRlbpYQyG$nCzIdp3HfmVTxy...` in the provided setup).

(E6): [disk_encryption_leak] [10 human-minutes]: Recover AES key from kernel.

Preparation: Since preparing this proof-of-concept requires root, it is already prepared on the provided `lab86` machine. For more details on how to set it up on your own Loongson 3A5000 machine, refer to the `README.md` in the artifact.

Execution: Terminal 1:

```
ssh lab86 "cd disk_encryption_leak;
  ./attacker"
```

Terminal 2:

```
ssh lab86 "cd disk_encryption_leak;
  cd leak_primitive; ./primitive;
  killall attacker"
```

Make sure to terminate the attacker in terminal 1 once you are done.

Results: Leaked key matches the expected AES key used for disk encryption.

(E7): [fuzzer] [10 human-minutes + 15 compute-minutes]: Discover LoongLeak via fuzzing.

Preparation: All commands are executed in the `fuzzer` directory.

```
make fuzzer_sw fuzzer_hw
```

Execution:

1. Run the fuzzer on QEMU (on your device):

```
python3 fuzzer.py
```

2. Modify `QEMU=True` to `QEMU=False` in `fuzzer.py`.

3. Copy the required files to `lab86`:

```
scp fuzzer_hw fuzzer.py bitmap.py lab86:
```

4. Run the fuzzer on lab86:

```
ssh lab86 python3 fuzzer.py
```

5. Download the results:

```
ssh lab86 zip -r out_hardware out_hardware
scp lab86:out_hardware.zip .
ssh lab86 rm -rf out_hardware.zip out_hardware
unzip out_hardware.zip
```

6. Run the analysis:

```
python3 comparer.py
```

Results: Among others, `comparer.py` identifies semantic differences in floating-point or vector load instructions like `fld.s`, `fld.d`, and `vld`.

A.5 Version

This document is based on the LaTeX template for Artifact Evaluation V20231005. The submission, reviewing, and badging methodology followed for the evaluation of this artifact can be found at https://secartifacts.github.io/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Fuzzing Open-Source GPU Hardware with SIMT Program Generation

Zibo Gao[†§], Jie Wang[†§], Qihang Zhou[†§], Lixiao Shan[†§], Junjie Hu[†§], Xiaoqi Jia [†§], and Zhiqiang Lv[†§]
[†]Institute of Information Engineering, Chinese Academy of Sciences.
[§]School of Cyber Security, University of Chinese Academy of Sciences.
{gaozibo,wangjie2024,zhouqihang,shanlixiao,hujunjie,jiaxiaoqi,lvzhiqiang}@iie.ac.cn

A Artifact Appendix

A.1 Abstract

This artifact implements FuzzGPU, a GPU fuzzing framework for automated vulnerability discovery at the Register Transfer Level (RTL). It uses a generator to produce valid SIMT programs with diverse control and data flows, stressing GPU execution behavior. It also handles microarchitectural nondeterminism (e.g., caches and memory timing) and enables trace-based comparison against an ISA oracle for instruction-level bug localization. Evaluation shows FuzzGPU finds new RTL and ISS bugs. To the best of our knowledge, no publicly available GPU hardware fuzzer exists for comparison. We therefore adapted Cascade and DiveFuzz to GPUs as baselines that generate instruction-level data and control flows. Experimental results show that FuzzGPU achieves higher coverage and throughput than the GPU ports of Cascade and DiveFuzz. We also package the Devices Under Test (DUTs), including Vortex and Ventus OpenGPGPU. Overall, this artifact demonstrates FuzzGPU's functionality as presented in the paper and enables reproduction of the main results that motivate its design.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

FuzzGPU targets RTL designs and runs in simulation, and thus not compromises the host system executing the experiments. Broader ethical and safety considerations are discussed in the paper.

A.2.2 How to access

The artifact is available on Zenodo (https://doi.org/10.5281/zenodo.20321391) and can also be cloned via Git:

```
git clone --depth 1 --recursive \
https://github.com/cassuto/fuzzgpu.git
```

A.2.3 Hardware dependencies

To reproduce the performance results, we recommend using an Intel(R) Xeon(R) 6982P-C system with 192 vCPUs and 768 GB of RAM.

A.2.4 Software dependencies

Our artifact requires a Linux distribution, `make`, and `Docker v29.5.3` as prerequisites. The Dockerfile automatically installs the remaining dependencies. Experiments are designed to run inside the Docker container.

A.2.5 Benchmarks

FuzzGPU was evaluated on two open-source GPUs, Vortex and Ventus OpenGPGPU. We also compare its performance against GPU ports of two CPU fuzzers: Ported-Cascade and Ported-DiveFuzz.

A.3 Set-up

A.3.1 Installation

Clone the Git repository and build the Docker image:

```
make docker
```

This step performs clean installation. It downloads, builds, and installs all dependencies and configures the metadata. A stable Internet connection is required to avoid missing-file errors. Proxy options are provided in the Makefile and Dockerfile. This step may take up to 24 machine-hours and consume ~200 GB of disk space.

A.3.2 Basic Test

Run smoking tests:

```
scripts/run_smoking_tests.sh --all
```

The expected successful output is:

```
================================================
>>> SMOKING SUITE RESULT: PASS
>>> units_total=6 units_pass=6 units_fail=0
================================================
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): FuzzGPU significantly improves instruction completion rate. This is shown in Figure 10.

(C2): FuzzGPU generates memory relation edges, including *co*, *coe*, *rf*, *rfe*, *rfi*, *fr*, and *ppo*. This is shown in Table 2.

(C3): FuzzGPU achieves higher generation throughput than Ported-Cascade and Ported-DiveFuzz. FuzzGPU also achieves higher effective throughput than Ported-DiveFuzz and comparable performance to Cascade. This is shown in Figures 11 and 12.

(C4): The instruction generation time of FuzzGPU accounts for a smaller fraction than RTL emulation time in the per-test fuzzing time breakdown. This is shown in Figure 14.

(C5): FuzzGPU improves coverage over Ported-Cascade and Ported-DiveFuzz. This is shown in Figure 15.

(C6): Ablation shows that removing any component (generation stack, barrier planner, or taint propagation) reduces program metrics and coverage. This is shown in Section 5.4.

(C7): FuzzGPU finds most of the bugs very fast. This is shown in Figure 16.

A.4.2 Experiments

(E1): [5 human-minutes + 24 machine-hour + 200GB disk]: Build the Docker image, refer to A.3.
How to: Build image using the provided Dockerfile.
Preparation: None.
Execution: In the repository, execute `make docker`.
Results: The Docker image will ultimately be built successfully, as confirmed by Docker's output.

(E2): [5 human-minutes + 1 machine-hour]: Instruction completion rate (validating C1).
How to: Measure the instruction completion rate.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E2.sh`.
Results: `results/plots/instr_complete_rate.pdf` reproduces Figure 10.

(E3): [5 human-minutes + 8 machine-hour]: Throughput measurement (validating C3).
How to: Measure and compare the generation and effective throughput.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E3.sh`.
Results: `results/plots/gen_throughput_vortex_xlen64.pdf` reproduces Figure 11. `results/plots/effective_throughput_vortex_xlen64.pdf` reproduces Figure 12.

(E4): [5 human-minutes + 5 machine-hour]: Coverage measurement (validating C5).
How to: Measure the coverage.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E4.sh`.
Results: `results/plots/coverage_comparison_ventus_x32_combined.pdf` reproduces Figure 15.

(E5): [5 human-minutes + 2 machine-hour]: Ablation study (validating C6).
How to: Run the ablation study evaluation.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E5.sh`.
Results: It reproduces Section 5.4.

(E6): [5 human-minutes + 5 machine-hour]: Fuzzing campaign (validating C2, C4, and C7).
How to: Run fuzzing campaign.
Preparation: Upon completion of E1.
Execution: Execute: `scripts/run_E6.sh`.
Results: It reproduces Table 2, Figure 14 (`results/plots/time_breakdown_fuzzgpu.pdf`), and Figure 16 (`results/plots/bug_discover_time.pdf`).

A.5 Notes on Reusability

- We provide reduced-parameter guidance in `README.md`. In particular, this file explains how to scale down `scripts/run_E2.sh` – `scripts/run_E6.sh` by editing the top-level parameters in the corresponding `scripts/run_*.sh` files, such as the number of seeds, runtime limits, target architectures, methods, coverage types, and ablation configurations. These reduced runs are intended to complete faster while preserving the qualitative trends.

- For fuzzer extension, Doxygen generates API documentation at `docs/html/index.html`, covering the generator, architecture-support, and difftest interfaces. Using Graphviz `dot`, it also visualizes inheritance, collaboration, include, and directory graphs to help identify extension points for new RTL designs, ISS backends, and generator modules.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Injected and Leaked: Actively Inducing Side-Channel Leakage Using Electromagnetic Injection and Hardware Nonlinearity

Haoran Yan[1], Ziyu Shao[1], Shuhao Zhang[1], Qinhong Jiang[2], Yan Long[1]
[1]The Hong Kong University of Science and Technology (Guangzhou)
[2]The Hong Kong Polytechnic University

A Artifact Appendix

A.1 Abstract

This artifact supports the audio-eavesdropping case studies and the speech-enhancement pipeline of the paper. It comprises: (i) released case-study audio for cross-wall eavesdropping, closed-loop manipulation, and long-distance eavesdropping; (ii) the speech-enhancement code; (iii) a released model checkpoint; (iv) preprocessing, training, and inference scripts; and (v) released noise samples. The goal of the artifact is to enable evaluators to inspect the key results and claims described in the paper, including the released case stud audio evidence and the speech-enhancement pipeline that reconstructs intelligible speech from recovered eavesdropped audio. Consistent with the paper's responsible-release policy, security-sensitive operational details that could directly enable misuse, including vulnerable injection frequencies and closed-loop control logic, are intentionally omitted.

A.2 Description & Requirements

The artifact contains the public open-science materials focusing on released case-study audio recordings and the speech-enhancement codebase described in the paper. The artifact is organized as follows.

Case study audio/ contains the recordings for cross-wall eavesdropping, closed-loop manipulation, and long-distance eavesdropping described in Section 5.

`Cross-wall eavesdropping/` contains the cross-wall case-study audio. `Clean/` holds the clean source utterances (male and female versions) covering privacy- and confidentiality-sensitive content: a personal meeting location, a bank-card PIN and transfer amount, and a business price quote. The two `Eavesdropped/` subdirectories contain the corresponding speech recovered through the injection-induced EM side channel in a home/hotel-style private setting and a meeting-room business setting, supporting validation of the paper's cross-wall eavesdropping results.

`Closed-loop manipulation/` contains the audio materials for the landline-phone closed-loop manipulation case study. `Clean audio/` contains original audio generated directly through TTS. `Eavesdropped audio/` contains Alice's speech recovered through the injection-induced side channel. `Enhanced eavesdropped audio/` contains the recovered speech after applying the released speech-enhancement pipeline. `Synthesized audio/` contains adversarial speech generated to alter the conversation, such as changing a quote, reversing an agreement, or changing navigation instructions. `Audio recorded from Alice/` contains victim-side microphone recordings of Alice's speech. `Audio recorded from injection/` contains victim-side microphone recordings of the injected malicious speech. Together, these files contain audio samples across the eavesdrop-synthesize-inject chain described in the paper.

`Long distance eavesdropping/` contains the original audio generated directly through TTS and eavesdropped audio samples at 12 m, 18 m, 24 m, and 30 m, corresponding to the extended-range eavesdropping experiment enabled by higher-power EM injection.

Enhance Code/ contains the SGMSE-based speech-enhancement pipeline used to enhance recovered audio described in Section 3.4. It includes preprocessing, training, and enhancement scripts; the model implementation; package requirements; the training configuration; released noise samples; and a released checkpoint with sensitive textual metadata.

The top-level `README.md` is the primary guide for artifact evaluation. It documents the artifact scope, directory layout, external dataset provenance, and commands for exercising the speech-enhancement pipeline.

A.2.1 Security, privacy, and ethical concerns

Consistent with the paper's responsible-release policy, this artifact is carefully scoped to avoid disclosing security-sensitive operational details that could directly facilitate misuse, including vulnerable injection frequencies and active injection control logic.

As with any software artifact, inspectors or users should exercise ordinary caution when installing dependencies and loading machine-learning checkpoints, as these actions involve the standard risks associated with executing third-party code. Overall, the released materials are intended to enable artifact evaluation and evidence inspection while preserving the responsible-disclosure boundary of the work.

A.2.2 How to access

The artifact is publicly archived on Zenodo at the concept DOI https://doi.org/10.5281/zenodo.18500378. The project website containing demo video is available at https://injecteave.github.io/.

A.2.3 Hardware dependencies

The authors successfully ran the released speech-enhancement pipeline on NVIDIA A800 and GeForce RTX 4090 GPUs. For practical inference and optional training, we recommend an NVIDIA CUDA-capable GPU with more than 16 GB of GPU memory.

A.2.4 Software dependencies

The pipeline targets Python 3.8+ with CUDA 11.8+ recommended. Python package requirements are listed in `Enhance Code/requirements.txt` (including `torch`, `torchaudio`, `librosa`, `soundfile`, `pytorch-lightning`, and related packages) and can be installed with a single `pip` command in A.3.1.

A.2.5 Benchmarks

The main evaluation workloads are the released case-study audio files under `Case study audio/`, including cross-wall eavesdropping audio, closed-loop manipulation audio, and long-distance eavesdropping audio. These files support evaluators to inspect the released evidence and compare clean reference audio with corresponding recovered, enhanced, synthesized, or victim-side recorded audio where applicable.

`Enhance Code/checkpoints/model.ckpt` contains the trained speech-enhancement model, allowing evaluators to run inference directly on released eavesdropped audio without retraining. The released noise samples (`Enhance Code/data/noise/`, 118 WAV files) support the documented preprocessing pipeline and future reuse of the data-generation workflow.

The clean-speech source for constructing paired enhancement training data is the LibriSpeech `train-clean-100` subset (OpenSLR resource SLR12, archive `train-clean-100.tar.gz`, CC BY 4.0), available from http://www.openslr.org/12/. LibriSpeech provides clean speech utterances used as ground-truth targets when synthesizing clean/noisy training pairs. It is needed for optional preprocessing, training, or reuse experiments.

A.3 Set-up

A.3.1 Installation

Download and extract the artifact from the Zenodo DOI https://doi.org/10.5281/zenodo.20432240. All commands are run in the `Enhance Code/` directory. Install the Python dependencies:

```
cd "Enhance Code"
python -m pip install -r requirements.txt
```

A.3.2 Basic Test

Run a single-file inference on one released eavesdropped recording inside `Enhance Code/` after installation:

```
cd "Enhance Code"
python enhancement.py \
  --test_file "../Case study audio/Closed-loop manipulation
    ↪ /Eavesdropped audio/Yes, turn right and you will
    ↪ see the hotel.wav" \
  --enhanced_dir enhanced_audio \
  --ckpt checkpoints/model.ckpt \
  --N 30 --device cuda
```

Each path is a single logical line; a hooked arrow (↪) marks only visual wrapping within the column, so no file name is broken across a line continuation.

Expected behavior: the script loads the checkpoint (`Model loaded`), writes one enhanced WAV file under `enhanced_audio/`, prints the saved output path (`Enhanced file saved to: ...`), and finishes with `Enhancement completed! Output: enhanced_audio`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The artifact provides the released audio evidence supporting the paper's case studies in Section 5: cross-wall eavesdropping in Section 5.1, closed-loop landline-phone manipulation in Section 5.2, and long-distance eavesdropping in Section 5.3. The released files are organized so that clean reference audio can be directly inspected with corresponding eavesdropped, enhanced, synthesized, or victim-side recorded audio where applicable.* This is validated by experiment (E1).

(C2): *The artifact provides the SGMSE-based speech-enhancement pipeline described in Section 3.4. Using the released checkpoint, evaluators can run inference on a released eavesdropped audio sample and obtain an enhanced output, corresponding to the paper's claim that enhancement improves recovered audio intelligibility.* This is validated by experiment (E2).

A.4.2 Experiments

(E1): *Inspect released layout and audio* [15 human-minutes + negligible compute + <1 GB disk]: confirm the released audio evidence is present and well-formed.
How to: Inspect the directory tree and audio with any local audio tool.
Preparation: Extract the artifact.
Execution: verify the three case-study groups. Inspect one released audio file with any local audio tool.
Results: The directory structure matches the artifact README, and the audio plays successfully, supporting (C1).

(E2): *Speech-enhancement inference* [15 human-minutes + ∼minutes compute on GPU + <1 GB disk]: enhance a released eavesdropped recording with the released checkpoint.
How to: Run the Basic Test inference command.
Preparation: Complete Set-up (Installation). A CUDA GPU is recommended; CPU execution has not been validated by the authors and may be slow or fail depending on PyTorch/checkpoint loading.
Execution: From `Enhance Code/`, run `enhancement.py` with `--ckpt checkpoints/model.ckpt` on a file from `Case study audio/Closed-loop manipulation/Eavesdropped audio/`.
Results: One enhanced WAV file is written to `enhanced_audio/`, and the completion message is printed. The enhanced output can be compared against the corresponding file in `Closed-loop manipulation/Enhanced eavesdropped audio/`, supporting (C2).

A.5 Notes on Reusability

The speech-enhancement component is designed to be reusable beyond the paper's case studies. Because the model is trained on paired data built from public clean speech and released noise samples—rather than device-specific recordings—it learns general distortion and noise patterns instead of memorizing a particular device, speaker, or environment. As discussed in Section 3.4, this yields strong generalization across injection-induced side-channel settings: the same checkpoint applies to recovered audio from different devices and scenarios without device-specific retraining, and it may also have value as a general-purpose speech-enhancement and audio-denoising component outside this context.

The released code exposes separate preprocessing, training, and inference entry points. Users can apply the released checkpoint directly to new noisy audio with `enhancement.py`, build new paired data with `preprocess.py` from their own clean-speech and noise sources, or fine-tune the model on a small amount of domain-specific paired data to adapt it to a new acoustic environment or sensing modality.

The preprocessing and training scripts are provided as illustrative and reusable workflow components. They document how paired enhancement data can be constructed and how the model can be trained or fine-tuned for future use, but they are not intended to regenerate the exact released checkpoint. The training settings (model, STFT, SDE, and optimization) are documented in `configs/train_config.yaml` for traceable adaptation.

Optional data-construction and training checks

The following optional checks exercise the data-construction and training code paths and require downloading an external dataset. They are intended to illustrate and reuse the preprocessing/training workflow, not to reproduce the exact training set or regenerate the released checkpoint.

- *Preprocessing* [30 human-minutes + ∼tens of minutes compute + a few GB disk]: download `train-clean-100.tar.gz` from OpenSLR SLR12, then run `preprocess.py` on a small subset with the released `./data/noise` as `--noise_dir` and the SNR range documented in the artifact README (`--snr_min -15 --snr_max 15`). Paired `clean`/`noisy` files appear under the chosen `--output_dir`, exercising the nonlinear-simulation data pipeline.
- *Training*: `train.py`, configured through `configs/train_config.yaml`, is provided for completeness. It is not intended to reproduce the paper's training run or regenerate the released checkpoint.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: *DRVFuzz*: Data-Sensitive RISC-V CPU Fuzzing

Zehong Yu
Tsinghua University

Yuanliang Chen
Renmin University of China

Zhen Yan
Tsinghua University

Xudong Zhang
Tsinghua University

Zhensheng Xian
Tsinghua University

Yu Jiang
Tsinghua University

A Artifact Appendix

A.1 Abstract

We provide the source code of *DRVFuzz* along with a Docker-based evaluation setup to facilitate reproducing the results presented in our paper and to support *DRVFuzz*'s deployment for evaluating future RISC-V CPUs. These artifacts demonstrate that *DRVFuzz* is a fully functional data-sensitive CPU fuzzing tool capable of generating SDModel-guided test programs, executing differential fuzzing campaigns, and comparing coverage and transition-count results against existing RISC-V fuzzers.

A.2 Description & Requirements

Artifact contents. The artifact contains the following components.

- *DRVFuzz* source code, including the SDModel sensitive-data generator, execution-state labeler, transition extractor, and transition-guided seed feedback.
- Bundled simulator binaries and a RISC-V GNU toolchain used by the Docker image.
- A `README.md` with quick-start commands and and manual examples.

A.2.1 Security, privacy, and ethical concerns

DRVFuzz is a local RISC-V CPU fuzzing framework that generates and executes test programs on simulator-based CPU models. The artifact runs fuzzing workloads inside Docker and does not access production systems or third-party services. We recommend running it in an isolated container or VM. The reported bugs were handled through coordinated disclosure.

A.2.2 How to access

Our artifacts are accessible for permanent access on Zenodo at https://doi.org/10.5281/zenodo.20343558. Alternatively, they can be accessed via GitHub at https://github.com/YzhDDDing/DRVFuzz.

A.2.3 Hardware dependencies

We recommend an x86_64 Linux host with at least 8 CPU cores, 32 GiB RAM, and 100 GiB free disk space for reduced experiments. Full 24-hour campaigns benefit from 200 GiB free disk space and a CPU comparable to the paper's evaluation machine. macOS hosts can use Docker Desktop; Apple Silicon users should run the container with `-platform linux/amd64`.

A.2.4 Software dependencies

The host needs Docker, `unzip`, and standard shell utilities. The Docker image contains the remaining dependencies, including Python packages, the RISC-V GNU toolchain, *DRVFuzz*, and simulator binaries. Internet access is needed only to build the image, unless a prebuilt image is provided.

A.2.5 Benchmarks

The experiments require six open-source RISC-V CPU implementations as benchmark targets: BOOM-V3, BOOM-V4, Rocket, CVA6, Kronos, and Srv32. The artifact includes their RTL sources, simulation environments, and CPU configuration files.

A.3 Set-up

A.3.1 Installation

Unpack the artifact file and build the Docker image:

```
unzip DRVFuzz_artifact.zip
cd DRVFuzz_Impl
docker build -t drvfuzz-ae:latest .
```

If Docker cannot resolve package mirrors, build with host networking:

```
docker build --network=host -t drvfuzz-ae:latest .
```

Start an interactive container and mount a host output directory:

```
mkdir -p runs
docker run --rm -it -v "$PWD/runs:/runs" \
  drvfuzz-ae:latest
```

On Apple Silicon macOS hosts, use:

```
docker run --platform linux/amd64 --rm -it \
  -v "$PWD/runs:/runs" drvfuzz-ae:latest
```

A.3.2 Basic Test

Inside the container, run:

```
DRVFuzz --help
DRVFuzz diff \
  --testcase-config configs/rv64.toml \
  --riscv-impl spike cva6 \
  --run-dir /runs/smoke_cva6_spike \
  --impl-timeout configs/timeout.toml \
  --run-options configs/run_options.toml \
  --concurrency 1 \
  --max-runs 5 \
  --sdmodel \
  --transition-guided
```

The expected result is a successful short differential-fuzzing run under `/runs/smoke_cva6_spike`. The directory contains the generated programs, Spike and CVA6 execution logs, and SDModel transition reports. If a mismatch is found during the smoke test, *DRVFuzz* also writes the corresponding failure report and reproducer.

A.4 Evaluation workflow

A.4.1 Major Claims

C1. *DRVFuzz* detects data-sensitive bugs in real-world RISC-V CPUs. In the paper, *DRVFuzz* found 22 previously unknown bugs across BOOM-V3, BOOM-V4, Rocket, CVA6, Kronos, and Srv32; 20 were assigned CVEs. This claim is supported by Tables 2 and 3.

C2. *DRVFuzz* achieves higher microarchitectural coverage than Cascade and DiveFuzz on the six evaluated CPUs. This claim is supported by Figure 8.

C3. *DRVFuzz* explores more data-sensitive execution-state transitions than Cascade and DiveFuzz. This claim is supported by Table 4, which reports average unique transitions per 1000 instructions.

C4. Both components, SDModel and transition-guided fuzzing, contribute to bug-finding effectiveness. This claim is supported by the ablation study in Figure 9.

A.4.2 Experiments

(E1): *[Build and smoke test.] [5 human minutes, 2 machine hours] (Optional):*
How to: follow the installation and basic functionality test above.
Preparation: Build the Docker environment and ensure that the required dependencies, RISC-V toolchains, simulators, and benchmark configurations are available.
Execution: Run the basic functionality test with a short SDModel-enabled, transition-guided CVA6-vs-Spike differential fuzzing campaign.
Results: Docker build succeeds, the CLI prints its help message, and a short SDModel-enabled, transition-guided CVA6-vs-Spike differential-fuzzing run completes with per-run logs and transition reports.

(E2): [Bug-finding campaign.] [10 human minutes, 24 machine hours]:
How to: run *DRVFuzz* differential fuzzing with SDModel and transition-guided feedback. For example, for Rocket:

```
DRVFuzz diff
--testcase-config configs/rv64.toml
--riscv-impl spike rocket
--run-dir /runs/bug_rocket_full
--impl-timeout configs/timeout.toml
--run-options configs/run_options.toml
--concurrency 1
--sdmodel
--transition-guided
```

For BOOM-V3, BOOM-V4, and CVA6, replace `rocket` with `boom-v3`, `boom-v4`, or `cva6`. For Kronos and Srv32, use `configs/rv32.toml` and replace the implementation name with `kronos` or `srv32`.
Preparation: Select the target RISC-V CPU benchmark and prepare the corresponding configuration files. Ensure that the CPU simulator and Spike reference model are correctly configured for differential testing.
Execution: Launch DRVFuzz with SDModel and transition-guided fuzzing enabled. The campaign generates test programs, executes them on both the target CPU and Spike, and detects architectural mismatches.
Results: If a mismatch is found, DRVFuzz creates a per-run directory containing logs, generated assembly, execution outputs, and minimized reproducers. Full bug rediscovery is stochastic; the expected paper-scale outcome is the trend reported in Table 3, where DRVFuzz detects substantially more bugs than Cascade and DiveFuzz.

(E3): *[Coverage comparison.] [5 human minutes, 24 machine hours]:*
How to: `run_coverage_compare /runs/coverage_24h`
Preparation: Prepare the coverage comparison environment and ensure that the required benchmark results

from the fuzzing campaigns are available.
Execution: Run the coverage comparison script to collect branch and toggle coverage results for the evaluated RISC-V CPUs.
Results: Each target directory contains coverage results, which can be further used to plot coverage trends. The expected paper-scale result is the same trend as Figure 8: DRVFuzz achieves the highest final branch and toggle coverage on the evaluated targets.

(E4): *[Transition-count comparison.] [5 human minutes, 1 machine hour]:*
How to: `run_transition_count_compare`
Preparation: Prepare the transition-count evaluation environment and collect the normalized execution inputs required for transition analysis.
Execution: Run the transition-count comparison script to measure the number of unique transitions triggered by different fuzzing approaches.
Results: The expected paper-scale trend is that DRVFuzz achieves the largest unique-transition density on every target, matching Table 4.

(E5): *[Component ablation.] [10 human minutes, 24 machine hours]:*
How to: Use the same differential fuzzing command as E2 while toggling the exposed flags. The full *DRVFuzz* configuration enables both `-sdmodel` and `-transition-guided`. The $DRVFuzz_{m^-}$ variant omits `-sdmodel`; the $DRVFuzz_{g^-}$ variant uses `-sdmodel` but omits `-transition-guided`; the $DRVFuzz_{\text{mode}}$ variant uses `-sdmodel -mode-guided`.
Preparation: Prepare the same benchmark configurations and execution environment used in E2. Configure the desired DRVFuzz variant by enabling or disabling the corresponding command-line flags.
Execution: Run differential fuzzing campaigns for each DRVFuzz variant under identical experimental settings.
Results: Compare the number of detected bugs and detection time among different configurations. The full configuration is expected to find bugs faster and more frequently than the variants, consistent with Figure 9.

A.5 Notes on Reusability

The artifact is designed so that new RISC-V CPUs can be integrated by adding a simulator wrapper, target configuration, memory layout, and trace extraction interface. The SDModel and transition-count logic are shared across targets. The comparison scripts intentionally preserve generated seeds, logs, transition reports, and coverage histories under `/runs` so that evaluators can inspect or replay individual samples.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: "Tap" Without Tapping: A Tag Discovery Forgery Attack on Android NFC

Yilin Li
School of Cyber Science and Technology, Shandong University

Jianliang Wu(✉)
Simon Fraser University

Chaoshun Zuo
Independent Researcher

Qingchuan Zhao
City University of Hong Kong

Xiaofeng Liu(✉) Xiangpu Song Chengyu Hu Shanqing Guo(✉)
School of Cyber Science and Technology, Shandong University

A Artifact Appendix

A.1 Abstract

This artifact accompanies our USENIX Security '26 paper *"Tap" Without Tapping: A Tag Discovery Forgery Attack on Android NFC*. The artifact has been awarded the *Artifacts Available*, *Artifacts Functional*, and *Results Reproduced* badges. The reproduced-results scope is limited to the released static-analysis pipeline and aggregate-measurement results.

The package provides a reviewer-runnable measurement and reproduction workflow. The main path uses shipped intermediate records to reproduce the paper-level aggregate numbers and derived CSV/Markdown outputs without re-parsing the full 76,333-app corpus. The artifact also includes controlled toy APKs and qualitative case-study evidence for functionality inspection. Real-world attacker source code, app-specific payloads, vendor-specific trigger recipes, raw APKs, and AndroZoo credentials are not included.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The main evaluation path is offline and non-destructive. It runs local scripts over shipped intermediate data and derived result files. It does not contact third-party services, target real-world applications, modify system settings, or require an Android device.

The package includes controlled demonstration APKs only. These APKs demonstrate the dispatch-forgery pattern in a toy setting and do not target third-party applications or live services. The artifact withholds real-world attacker implementations, app-specific payloads, vendor-specific exploit recipes, and reusable end-to-end attack instructions. The full boundary is documented in `WITHHELD_MATERIALS.md`.

Optional Android-side demonstrations should be run only in an isolated emulator or non-personal test device. These optional paths are not required for the main functionality and reproduction workflows covered by the awarded badges.

A.2.2 How to access

The final artifact is archived on Zenodo under the following permanent concept DOI:

`https://doi.org/10.5281/zenodo.20303896`

This concept DOI is the permanent stable reference for the artifact and resolves to the latest archived version. The artifact remains under restricted access because it contains sensitive per-app measurement and validation materials while platform-side remediation and coordinated disclosure remain relevant. Access requests can be submitted through the Zenodo record. The package includes a `README.md` with installation, evaluation, and reproduction instructions.

A.2.3 Hardware dependencies

None for the main evaluation path. The artifact runs on a standard CPU-based host. It does not require GPUs, TEEs, FPGAs, embedded boards, high-memory servers, Android devices, raw APK downloads, AndroZoo API keys, or access to third-party services.

An Android device or emulator is needed only for optional device-side demonstrations or dynamic re-collection. These optional paths are outside the byte-for-byte reproduction claim.

A.2.4 Software dependencies

The default path requires a POSIX-like environment. The author-tested and officially supported range is Python 3.9–3.12. Newer Python versions may also work, but are not part of the author-tested support range. The script `setup.sh` installs the required Python packages, including `pandas`, `loguru`, `tqdm`, `requests`, and `tabulate`. The `adb` tool is needed only for optional Android-side demonstrations.

Windows users should use WSL2 or an equivalent POSIX-compatible environment.

A.2.5 Benchmarks

The dataset is a custom AndroZoo-derived corpus. The artifact ships the SHA list and the precomputed intermediate records needed for reproduction. Raw APKs and AndroZoo credentials are not redistributed.

A.3 Set-up

A.3.1 Installation

From the artifact root, run:

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

This installs the Python dependencies needed for the reviewer-runnable path.

A.3.2 Basic Test

After installation, run:

```
$ ./check_artifact.sh
$ ./run_smoke_test.sh
```

The first script checks required files and package integrity. The second script runs a short sanity test over the packaged data and derived outputs. A successful run should finish without errors.

For full aggregate reproduction, run:

```
$ ./run_reproduce.sh
```

A successful execution ends with:

```
RESULT-A: PAPER-CLAIM CHECK PASS
RESULT-B: BYTE-FOR-BYTE DIFF PASS
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Dataset construction. The paper's analyzable set contains **76,333** apps, derived from 76,345 candidate SHA entries and 76,334 manifest-feature records. This is checked in experiment E1.

(C2): Reader/Writer NFC apps and handlers. The analysis identifies **928** apps with Reader/Writer NFC handlers and **1,089** Reader/Writer handler components. This is checked in E1.

(C3): Potentially vulnerable apps and handlers. The analysis identifies **601** apps with at least one potentially vulnerable handler and **703** unique potentially vulnerable handler components. This is checked in E1.

(C4): Dynamic-validation candidates. The dynamic-validation input contains **708** attempt-level rows. This is checked in E1.

(C5): Dynamic triggerability. The 708 attempted invocations yield 618 successful installs and **550** `INVOKE_SUCCESS` outcomes. This is checked in E1.

(C6): Controlled dispatch-forgery demonstration. The controlled example APKs demonstrate the forged NFC dispatch pattern in a toy setting. This is inspected in E3 and is not part of the byte-for-byte aggregate reproduction claim.

A.4.2 Experiments

(E1): Static-pipeline and aggregate reproduction [5 human-minutes + 1–3 compute-minutes + 1GB disk].
This is the main reproduction experiment corresponding to the *Results Reproduced* badge. It supports C1–C5.
Preparation: Run `./setup.sh` if needed, then run `./check_artifact.sh`.
Execution: Run:
`$ ./run_reproduce.sh`
Results: The script regenerates a fresh reproduction output tree from the shipped intermediates, checks the headline paper claims, and byte-compares regenerated CSV/Markdown outputs against the shipped copies. The expected final output is:
`RESULT-A: PAPER-CLAIM CHECK PASS`
`RESULT-B: BYTE-FOR-BYTE DIFF PASS`
The reproduced dynamic-success total is 550.

(E2): Smoke test [2 human-minutes + under 1 compute-minute + under 1GB disk].
This experiment demonstrates the functionality corresponding to the *Artifacts Functional* badge.
Preparation: Run `./setup.sh` if needed.
Execution: Run:
`$ ./run_smoke_test.sh`
Results: The expected result is a successful sanity test over the packaged data and derived outputs.

(E3): Controlled dispatch-forgery demonstration [10–20 human-minutes + optional device/emulator time].
This experiment supports C6. It is optional and is not part of the *Results Reproduced* claim.
Preparation: Read `poc/README.md`. Use an isolated emulator or non-personal test device if running the Android-side demonstration.
Execution: Follow `poc/README.md`. The helper entry point is:
`$ cd poc`
`$ ./fire_attacker.sh`
Results: The expected result is a controlled toy demonstration of forged NFC dispatch between the included example APKs. It does not target third-party applications or live services.

(E4): Optional APK-level static demo [5–10 human-minutes + input-dependent compute time].
This experiment helps reviewers inspect the static

pipeline on APK inputs. It is not required for paper-level aggregate reproduction.

Preparation: Use the default controlled APK for the single-APK path, or place APKs in the batch input directory.

Execution: Run:

```
$ ./run_single_apk_demo.sh
$ ./run_batch_apk_demo.sh
```

Results: The scripts show how the static pipeline emits feature, reachability, exposure, and validation-harness records. The paper-level aggregate reproduction uses the shipped intermediates rather than re-downloading or re-parsing the full corpus.

A.5 Notes on Reusability

The static side of the artifact can be reused with compatible feature and reachability inputs. The dynamic-aggregation side can consume CSV files with the same schema as the released validation-command or dynamic-result records. This lets future evaluators inspect the same pipeline structure on additional APK sets while keeping the released package safe and self-contained.

The artifact intentionally separates reusable measurement code from sensitive operational details. This supports independent inspection of the paper's static-analysis and aggregate-measurement claims without releasing real-world target-specific exploit materials during ongoing remediation.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: AutoFail: Breaking Web Boundaries using Android's Autofill Framework

Riccardo Lamarca
TU Wien

Philipp Beer
TU Wien

Marco Squarcina
TU Wien

A Artifact Appendix

A.1 Abstract

Our artifact includes: ADAPT, an automated differential testing framework for detecting autofill vulnerabilities in browsers and password managers; proof-of-concept applications demonstrating the Cross-Context Account Oracle attack and its mitigation; scripts and datasets for the real-world analysis of embeddability headers and iframe configurations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The web-crawler is set to spawn the browser without the sandbox enabled, which can expose the machine of the evaluator to security risks. We suggest running the crawler in a virtual machine or a disposable environment.

The proof-of-concept app of the Cross-Context Account Oracle attack is solely intended to demonstrate the attack and it does not collect any sensitive information.

A.2.2 How to access

The artifacts are hosted at https://github.com/SecPriv/autofail and archived at https://doi.org/10.5281/zenodo.20441978.

A.2.3 Hardware dependencies

System requirements: We recommend running ADAPT and the analysis tools on a machine with at least 16GB RAM. The APKs of the browsers and password managers and the Frida server require a device with arm64 architecture. We recommend either a physical Pixel 6a or an emulator. We provide the code to set up and run the emulator on macOS in the `emulator` directory.

A.2.4 Software dependencies

Host machine: We require a host machine running Linux or macOS. Note that macOS is required to use the code that we provide to set up the emulator.

Android image: We require Android 16 (API level 36) with root access (`adb` must be able to run as root) .

Google Cloud account: To execute E4, a Google Cloud account with access to BigQuery is required.

A.2.5 Benchmarks

Tranco Top 1M list: The crawler uses a pre-processed version of the Tranco top 1M websites list, included in the repository at `RealWorldAnalysis/IframeConfigurationAnalysis/crawler/data/`.

BigQuery HTTP Archive dataset: The embeddability analysis queries the public HTTP Archive dataset available on Google BigQuery. We provide the result of the query in `RealWorldAnalysis/HeaderConfigurationsAnalysis/bq_result_splitted/`

Pre-crawled iframe data: We provide pre-crawled iframe configuration data for 3,522 websites in `RealWorldAnalysis/IframeConfigurationAnalysis/crawler/data/bucketed_1M_out/` to enable immediate verification of results (E7).

A.3 Set-up

A.3.1 Installation

Clone the repository by running: `git clone https://github.com/SecPriv/autofail`.

Host machine dependencies: Run the `setup.sh` script to install the dependencies on the host machine.

Emulator: Run the `emulator/launch_emulator.sh` script to install and run the emulator. As an alternative, you can use a physical Android device.

Android device one time set-up: Run `emulator/install_apks.sh` to install the required APKs. Follow the one-time set-up in `ADAPT/README.md` to install the root CA certificate and configure domain resolution for `a.com` and `b.com`.

Run the server: Run `./ADAPT/run_server.sh` to perform the network set-up of the device, run Frida and start the server.

A.3.2 Basic Test

Run the `basic_test.sh` script that verifies that the host machine sees the Android device, that the device can reach the server and that the Frida server process is running.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The provided test suite (Table 2) reproduces the browser and password manager vulnerabilities summarized in Table 3.

(C2): Our proof-of-concept exploits demonstrate that the applications listed in Table 6 are vulnerable to the Cross-Context Account Oracle.

(C3): The proof-of-concept application implementing the Cross-Context Account Oracle mitigation (Section 7.2) is functional and effective.

(C4): The raw dataset of requests used in Section 8.1 (**Embeddability Analysis**) can be successfully retrieved.

(C5): Executing the analysis scripts on the Section 8.1 dataset reproduces the finding that 485,470 sites are embeddable, as well as the top 5 combinations of XFO and CSP headers reported in Table 7.

(C6): The web-crawler described in Section 8.2 (**Iframe Analysis**) can be executed to successfully collect the required iframe embedding data.

(C7): Running the provided evaluation scripts on the crawled iframe dataset reproduces the statistical findings detailed in Section 8.2 (**Iframe Analysis**), Table 8, and Table 9.

A.4.2 Experiments

(E1): [Test suite] [2 human-hours + 10 compute-minutes + 16GB disk]

Preparation: Navigate to the `ADAPT` directory. Connect the Android device. Install the specific browser and password manager versions detailed in Table 1. Following the **One time setup** instructions in `README.md`, install the root certificate authority required to sign the test suite's TLS certificates. For each password manager, manually create test credentials for both the `a.com` and `b.com` domains. Prior to testing a specific password manager, configure it as the device's preferred autofill service (e.g., via `Settings → Passwords, passkeys & accounts → Preferred service`). For Chrome and Brave, explicitly enable third-party external autofill within the respective browser settings.

Execution: Execute the `run_server.sh` script to initialize the test server. Proceed through the test cases by interacting with the server as outlined in the **Running the server** section of `README.md`.

Results: Observe the autofill behavior during the execution phase and verify that the results successfully reproduce the issues reported in Table 3 and detailed in Section 5.

(E2): [Cross-Context Account Oracle Attack] [10 human-minutes + 1 compute-minute + 16GB disk]

Preparation: Navigate to the `CrossContextAccountOracle` directory. Connect the Android test device (e.g., via ADB) and install the password managers specified in Table 6. For each password manager, manually create test credentials for the `a.com` domain. Prior to testing a specific password manager, configure it as the device's active autofill service (e.g., via `Settings → Passwords & passkeys & accounts → Preferred service`). Before executing the attack, open the password manager's main application interface and leave it running in the background.

Execution: Execute the `deploy-and-run.sh` script to compile, deploy, and launch the proof-of-concept attack application on the connected device.

Results: Observe the output displayed in the attack application's activity. Verify that the oracle successfully triggers and logs a result for the `a.com` domain, and confirm that no result is logged for the `b.com` domain.

(E3): [Cross-Context Account Oracle Mitigation] [10 human-minutes + 1 compute-minute + 16GB disk]

Preparation: Navigate to the `mitigation` directory. Connect the Android test device and execute the `install-apps.sh` script to install the required applications. Navigate to the device settings (e.g., `Settings → Passwords, passkeys & accounts → Preferred service`) and select `Secure Autofill PoC` as the active autofill service.

Execution: Execute the `run-poc.sh` script to launch the test environment. On the connected device, interact with the presented login form to trigger the autofill suggestion, and navigate the resulting security warning prompt.

Results: Observe the application's autofill behavior. Verify that the credentials are intentionally withheld initially, and are only listed as available *after* the user explicitly acknowledges the warning prompt.

(E4): [Headers dataset] [10 human-minutes + 10 compute-minutes + 1GB disk]

Preparation: Navigate to the `RealWorldAnalysis/HeaderConfigurationsAnalysis` directory. Ensure you have an active Google Cloud account with access to BigQuery.

Execution: Execute the query defined in `query.sql` via the BigQuery console (or CLI) and download the resulting dataset locally (e.g., as `results.json`). Next, execute the comparison script by passing the downloaded results file as an argument (e.g., `./compare_results.sh results.json`).

Results: Verify that the script outputs `SUCCESS`.

(E5): [Headers analysis] [1 human-minute + 1 compute-minute + 1GB disk]

Preparation: Navigate to the `RealWorldAnalysis/HeaderConfigurationsAnalysis` directory.
Execution: Execute the analysis script by running `verify_analysis.sh`.
Results: Verify that the script outputs `TRUE`.

(E6): [Crawler functional] [10 human-minutes + 20 compute-minutes + 1GB disk]
Preparation: Navigate to the `RealWorldAnalysis/IframeConfigurationAnalysis` directory.
Execution: Execute the crawler on a provided sample of websites by running `run_crawler_sample.sh`.
Results: Verify that the crawler completes its execution and confirm that the `evaluator_sample_output` directory is successfully generated and populated with the resulting data.

(E7): [Iframe Analysis] [10 human-minutes + 10 compute-minutes + 1GB disk]
Preparation: Navigate to the `RealWorldAnalysis/IframeConfigurationAnalysis` directory.
Execution: Execute the analysis script by running `verify_analysis.sh`.
Results: Verify that the script outputs `SUCCESS`.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Plain Text, Plain Risks: Measuring and Understanding HTTP Inclusion in Android WebViews at Scale

Philipp Beer, Sebastian Roth, Martina Lindorfer, and Marco Squarcina

A Artifact Appendix

A.1 Abstract

We provide multiple artifacts to reproduce the results presented in the paper and support future work that builds on our findings. Specifically, we include the following artifacts:

- **APK Download:** Scripts for downloading apps (`/apk-download`). Due to the dataset size, we cannot publicly release the full set of apps used. Reviewers are granted access to it as outlined in Section A.3.1.
- **HTTP Content Behavior Comparison:** Scripts to test the HTTP content behavior in WebViews and mobile browsers (`/http-content-comparison`).
- **Static Analysis Pipeline:** Code to run the static analysis, including the manifest analysis and the bytecode analysis (`/static-analyzer`).
- **Dynamic Analysis Pipeline:** Code to run the dynamic analysis, including the custom WebView provider and modified Monkey (`/ui-interactor` and `/webview-provider`).
- **Developer Survey:** The survey questionnaire (`/survey`).
- **Reproducibility Scripts:** Scripts to reproduce the results presented in the paper (`/reproducibility`).
- **Analysis Results:** (Intermediate) results (`/results`).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifacts accompanying the paper (excluding the results) are accessible at https://github.com/beerphilipp/plaintext-artifacts. The complete artifact is archived at https://doi.org/10.5281/zenodo.20393171.

A.2.3 Hardware dependencies

Access to the APKs. Due to the size of the APK dataset used in the paper, we cannot distribute it directly. Reviewers are granted access as outlined in Section A.3.1.

System Requirements. We require an Android emulator to run. As our APKs are built for ARM devices, we only support ARM architectures, specifically, a Mac with Apple Silicon. A minimum of 16 GB RAM and 250 GB available disk space are suggested.

A.2.4 Software dependencies

Operating System. We have tested and support macOS 26.5.

Docker. Install Docker (see https://docker.com/get-started).

Java. To be able to use the necessary Android dependencies, install a recent version of Java (see https://www.java.com/en/download/manual.jsp).

Android Dependencies. Install Android Studio (see https://developer.android.com/studio):

- Download Android Studio.
- Go to *Tools* > *SDK Manager* > *SDK Tools* and make sure *Android SDK Command-line Tools*, *Android Emulator*, and *Android SDK Platform-Tools* are installed. If not, install them.

A.2.5 Benchmarks

We require access to the APK dataset and the database snapshot (see Section A.3.1).

A.3 Set-up

A.3.1 Installation

Download the Artifact. Download the artifact from https://doi.org/10.5281/zenodo.20393171 and extract it in a directory of your choice.

Set Up Access to APK Dataset. To access the dataset, save the private SSH key that we provide for reviewers to a file named `~/.ssh/webview_key` and give it the correct permissions with `chmod 600 ~/.ssh/webview_key`. Researchers may request access to the dataset by contacting the authors.

Install Dependencies. See Section A.2.4 for instructions.

A.3.2 Basic Test

To verify correct installation and setup, first run `reproducibility/e1/start_emulator.sh` to set up and start an Android emulator. Then, run `reproducibility/basic_test.sh` in the repository's root directory. This command performs the following steps:

- Builds all required Docker images.
- Checks if the emulator is running.
- Checks if the SSH key is correctly saved.

The script should print *OK* to the console. This process may take a few minutes.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): HTTP Behavior of WebViews
WebView handles HTTP resources as described in the paper in terms of lack of auto-upgrading and blocking (cf. Section 3 and Table 1). This is proven by (E1).

(C2): Pipeline Functionality
The static analysis pipeline (cf. Section 4.2) is functional. This is proven by the experiment (E2).

(C3): Reproducibility of Paper Numbers
The measurement results in the paper (cf. Section 5) are reproducible. This is proven by experiment (E3).

A.4.2 Experiments

(E1): HTTP Content Behavior [30 human-minutes + 20 compute-minutes + 20 GB disk]
Preparation: Follow the steps in Section A.3.1 to set up the environment.
Execution: Run `reproducibility/e1/e1.sh` in the repository's root directory. The script builds the test app and starts the mixed content test server that is reachable via `mixed-content.test` on the Android emulator. Follow the output of the script to verify this experiment. To continue to the next (sub)experiment, press `Enter`, as prompted by the script.
Results: The server serves all resources over HTTP and HTTPS. Observe the device screen and whether HTTP content or HTTPS content was loaded. The script provides the expected outcome.

(E2): Analysis Pipeline [10 human-minutes + 20 compute-minutes + 20 GB disk]
Preparation: Follow the steps in Section A.3.1 to set up the environment.
Execution: Run `reproducibility/e2/e2.sh` in the repository. The script downloads a random subset of 10 apps from our dataset and runs the manifest and bytecode analyzer on these apps.
Results: The directory `reproducibility/e2/out` should contain subdirectories `bytecode` and `manifest`. The `bytecode` and `manifest` directories should contain 10 subdirectories corresponding to the APKs downloaded. For `bytecode`, each app directory should contain a result file for the bytecode analysis (`<package_name>.json`). For `manifest`, each app directory should contain the files `<package_name>-info.json`, `<package_name>-cleartext-config.json`, `<package_name>-fastbot-config.json`, and `<package_name>-manifest.xml`.

(E3): Measurement Results [1.5 human-hours + 5 compute-hours + 200 GB disk]
Preparation: Follow the steps in Section A.3.1 to set up the environment.
Execution: Run `reproducibility/e3/e3.sh <mongodb_snapshot> <gplay_db>` in the repository's root, where `<mongodb_snapshot>` refers to the path of the downloaded and extracted database snapshot and `<gplay_db>` to the path of the downloaded and extracted Google Play Crawl database. The script runs a MongoDB and Jupyter Notebook server in Docker. After the script finishes, open `http://localhost:8888` and run the notebooks (`*.ipynb`) in ascending order.
Results: Compare the results obtained from the notebooks with the results in the paper. Table 1 provides a mapping of notebooks to sections in the paper.

A.5 Notes on Reusability

To foster future research and make it easier for others to build on our work, we provide documentation in the `README.md` files included in each subdirectory.

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/usenixsec2026/.

Notebook	Notebook Cells	Paper Section	Notes
`1_generate_macros_dataset.ipynb`	Cells 2, 3	Section 4.1	
`2_generate_macros_success.ipynb`	Cells 2, 3, 4	Section 5.1	
`3_generate_macros_network_policy.ipynb`	Cells 2, 3	Section 5.2	
	Cells 4, 5, 8	Section 5.2, *Adoption over Time*	
	Cell 6	Section 5.2, *Allowed Domains*	
	Cell 7	Section 5.2, *Configuration Issues*	
`4_generate_cleartext_figure.ipynb`	Cell 4	Section 5.2, *Figure 2*	
`5_generate_macros_webview_usage.ipynb`	Cell 2	Section 5.3	*34,200* as per Section 5.1
	Cell 5	Section 5.3, *Figure 3*	
	Cell 7	Section 5.3, *Unique WebViews*	
	Cell 8	Section 5.3, *Origins of WebViews*	
`6_generate_macros_top_level.ipynb`	Cells 2, 3	Section 5.4, *Usage and Loaded Schemes*	
	Cells 3, 6	Section 5.4, *Local Address HTTP URLs*	
	Cell 4	Section 5.4, *Table 2*	
	Cell 5	Section 5.4, *Missing Protocol*	*21* may differ due to network changes
`7_generate_macros_enforcement_gaps.ipynb`	Cell 2	Section 5.5, *API Usage*	
	Cell 3	Section 5.5, *Commonly Used Base URLs*	
`8_generate_macros_policy_weakening.ipynb`	Cell 2	Section 5.6, *Prevalence*	
	Cell 5	Section 5.6, *Table 3*	
	Cell 6	Section 5.6, *Recorded Mixed Content*	
`9_generate_table_webview_capabilities.ipynb`	Cell 5	Section 5.7, *Table 4 + Risk*	*0.52%* located in caption of Table output, numbers in *Risk* extracted from Table
`10_generate_macros_loaded_content.ipynb`	Cell 2	Section 5.8	
	Cell 3	Section 5.8, *Resource Types*	only produces number for AIA Fetching
	Cell 8	Section 5.8, *Table 5 + Resource Types*	The numbers may differ, as Content-Type sniffing was performed (over the network) for URLs without an extension. For simplicity, the script skips this step, thus, the number in "Unknown" will be higher and for other resources, lower; numbers in *Resource Types* (except 80 AIA Fetching) extracted from Table
`11_generate_fyber_apps.ipynb`	Cell 2	Section 6.2, *Fyber SDK*	

Table 1: Mapping of evaluation notebooks and cells to main paper sections.



USENIX Security '26 Artifact Appendix: Leveraging Cryptographic Simulator Synthesis for Formally Verifying the FOO E-Voting Protocol

David Baelde
Univ Rennes, CNRS, IRISA

Adrien Koutsos
Inria

Justine Sauvage
Inria

A Artifact Appendix

This appendix describes the artifacts associated to the paper *Leveraging Cryptographic Simulator Synthesis for Formally Verifying the FOO E-Voting Protocol*, and explains how to reproduce the experiments presented in that paper.

A.1 Abstract

The artifacts are:

- the source code of the Squirrel proof assistant with our modified version of the `crypto` tactic;
- the Squirrel proof scripts for the paper's running example and case studies;
- the Squirrel proof scripts for the proof of vote privacy for the FOO e-voting protocol;
- static HTML files allowing to follow Squirrel runs on all proof scripts, step by step, without having to build and run the tool.

A.2 Description & Requirements

All files are available at the following URL:

`https://doi.org/10.5281/zenodo.17880702`

External software dependencies are freely available online, mostly on standard package managers.

Our code should run on any modern machine. It has been tested on standard laptops with Intel i7 and AMD Ryzen 7 CPUs with 30Go RAM. It should build and run on any standard Linux or MacOS distribution; it has been tested on Debian 13, Ubuntu 20.04.6 LTS, MacOS 26.2 and 15.7.3.

The artifacts have been integrated into the main line of development of the Squirrel proof assistant, i.e. the `master` branch of Squirrel's public repository[1]. We expect that our contributions will remain available in that repository, where they will be maintained and kept compatible with future evolutions of both Squirrel and its external dependencies. If one does not need to work with the exact archived version of our artifacts, it might become more practical to access and experiment with our contributions through the Squirrel repository. Note that the present document describes the software dependencies and build instructions for the archived artifact only; one should refer to the `README.md` file in Squirrel's repository for up-to-date information on how to build the current system.

[1] `https://github.com/squirrel-prover/squirrel-prover/`

A.2.1 Security, privacy, and ethical concerns

We do not identify any risk associated to building and running Squirrel on proof scripts. Squirrel itself does not modify or create files, and it does not communicate on the network.

A.2.2 Hardware dependencies

None.

A.2.3 Software dependencies

Our code relies on the following software (the provided versions are known to work, but slightly different versions should also work unless otherwise stated):

- `opam` 2.3.0 (any version above 2.0.0 should work)
- `ocaml` 5.4.0 (any version above 5.0.0 should work)
- `dune` 3.21.0
- `fmt` 0.11.0
- `ocamlgraph` 2.2.0
- `alcotest` 1.9.1
- `sedlex` 3.7
- `ppx_inline_test` v0.17.1
- `menhir` 20250912
- `menhirLib` 20250912

- `conf-which` 1
- `zarith` 1.14
- `why3` 1.8.2 (other versions are likely to break)
- `cvc5` 1.0.8 (other versions may not work)
- `z3` 4.12.2 or 4.13.2 (other versions may not work)

Most of these packages are installed automatically (with appropriate versions) using opam and a provided opam configuration file, as explained below.

Squirrel is generally used with Emacs Proof General to develop proof scripts, but can be used as a standalone tool to verify existing scripts. Therefore, this appendix does not include dependencies and instructions for Proof General. The interested reader may find details on how to setup Proof General with Squirrel in the `squirrel/README.md` file.

A.2.4 Benchmarks

Experimental results from the paper only rely on the proof scripts contained in the artifact.

A.3 Set-up

1. Install `opam` using a standard package manager, e.g. by running `apt-get install opam`.
2. Run `opam init` to set it up.
3. Run `opam switch create squirrel 5.4.0` to create a Squirrel switch based on OCaml 5.4.0.
4. Run `eval $(opam env --switch squirrel)` to sync your environment with the switch.

A.3.1 Installation

First install SMT solvers and Why3:

1. Run `opam install why3` to obtain the optional why3 package.
2. Get the binaries for CVC5 and Z3 from the official websites https://github.com/cvc5/cvc5/releases/tag/cvc5-1.0.8 and https://github.com/Z3Prover/z3/releases/tag/z3-4.12.2.

   Make sure the binaries are executable and available in your PATH under the names `cvc5` and `z3` — they may simply be copied to any directory in the path, as they do not rely on installed files.
3. Configure Why3 by running `why3 config detect`. Make sure CVC5 and Z3 are found.

Then, install dependencies and build Squirrel by running the next commands from Squirrel's source code directory, e.g. the `squirrel/` subdirectory of the artifact:

1. Run `opam install . -y --deps-only` to install opam packages required for Squirrel;
2. Run `make` to build Squirrel.

A.3.2 Basic Test

From the toplevel directory of the artifact, running `squirrel/squirrel` should display a short help message.

Running `squirrel/squirrel proofs/motivating.sp` should display a long output ending with "Goodbye!" before exiting with error code 0 in a few seconds.

To test that Squirrel is fully functional with SMT support, run `squirrel/squirrel proofs/kdf.sp`. It should display a long output ending with "Goodbye!" as before, in less than 30 seconds.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The improved crypto tactic included in our version of Squirrel runs as intended, implicitly synthesizing simulators with memoization. This is demonstrated in the Squirrel proof scripts `proofs/*sp`.

(C2): Our modified version of Squirrel allows to prove vote privacy for FOO. This is demonstrated by the Squirrel proof script `proofs/foo/main.sp` which relies on other scripts in the same subdirectory.

(C3): The execution times for these files are reasonable, as shown in Table 1 of our paper.

A.4.2 Experiments

(E1): Run Squirrel on all files for **(C1)** and **(C2)** to check that the proof scripts work as intended, establishing the claimed results.

This experiment takes less than 1 human-minute to launch and verify. It should take less than 5 computer-minutes.

How to: A Makefile is available to run Squirrel on all files. Alternatively, the testers may run Squirrel manually as shown above on individual files.

Preparation: Switch to the `proofs` subdirectory.

Execution: Run `make`.

Results: The output should show OK for each file.

(E2): Execution times for individual files may be recomputed by running Squirrel manually and timing it using the `time` utility.

This can be done easily, as described below, for files in `proofs/*.sp`. For files in `proofs/foo/*.sp`, one first needs to manually change the first lines

by replacing each `include <module>` directive by `include[admit] <module>`. This is to avoid that Squirrel checks the proofs in the included modules, to avoid counting their execution times multiple times.
Preparation: Switch to the `proofs` subdirectory.
Execution: Run the following command for each file: `time ../squirrel/squirrel <file.sp>`.
Results: The execution time is displayed.

(E3): Execution times of individual calls to the `crypto` tactic may be obtained by using Squirrel's internal `time` utility. To do so, occurrences of the `crypto` tactic in the files should be prefixed by `time`. For instance, in `ccapk1.sp` there are two calls to `crypto` as reported in Table 1, which should be modified as shown in Figure 1.
The execution time of each instrumented tactic will be shown at the end of the tactic's output. That output can be obtained by running Squirrel in interactive mode from Emacs' Proof General, and jumping to the relevant position in the proof script. Timing outputs can also be collected in batch using only the standalone tool as shown in Figure 1 for our example file.

A.5 Notes on Reusability

As mentioned above, our artifacts have been integrated in Squirrel's official repository, to ensure that they are maintained in the future and to facilitate their use e.g. as benchmarks for future works.

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/usenixsec2026/.

```
$ grep -n -C 3 "time crypto" foo/ccapk1.sp
194-  [happens(t)] -> equiv(frame@t).
195-Proof.
196- intro *.
197: time crypto ~no_subgoal_on_failure ~time_sensitive CCA2 (key:sk_mix1).
198- - smt ~no_macros.
199- - smt ~no_macros.
200- - smt ~no_macros.
--
351-[happens(t)] -> equiv(frame@t).
352-Proof.
353- intro *.
354: time crypto ~no_subgoal_on_failure ~time_sensitive CCA2 (key:sk_mix1).
355- - smt ~no_macros.
356- - smt ~no_macros.
357- - smt ~no_macros.

$ ../squirrel/squirrel foo/ccapk1.sp 2> /dev/null | grep time:
[dbg>time: 3.917882 <][goal> Focused goal (1/26):
[dbg>time: 4.021414 <][goal> Focused goal (1/26):
```

Figure 1: Instrumentation and measure of individual tactics



USENIX Security '26 Artifact Appendix: OS-SANITIZER: System-wide Latent Defect Inference in Linux Applications

Addison Crump
addison.crump@cispa.de
CISPA

Sahil Sihag
sahil.sihag@cispa.de
CISPA

Florian Bauckholt
florian.bauckholt@cispa.de
CISPA

Keno Hassler
keno.hassler@cispa.de
CISPA

Thorsten Holz
thorsten.holz@mpi-sp.org
MPI-SP

A Artifact Appendix

A.1 Abstract

Dynamic analysis is used throughout empirical testing, but primarily involves the detection of defects through the observation of corresponding faults; that is, something unwanted must happen for the defect to be detected. OS-SANITIZER is an eBPF-based tool for the passive detection of latent Linux application defects, observing events which indicate the capacity for a fault to manifest. Our prototype was evaluated quantitatively by measuring its performance in a benchmarking context and analyzing the report volume in a long-term deployment, and qualitatively by means of reimplementing existing strategies and actually using the tool to find defects in real software. Our provided artifact provides the prototype itself (including the reimplementation of existing work), experimental setup scripts, evaluation targets (i.e., the microbenchmarks and the scripts with which to measure performance on microbenchmarks, SPEC CPU 2017, and Browserbench), evaluation data of our own performance measurements, and the analysis scripts to filter and convert this data into LaTeX tables. We aim for all artifact evaluation badges based on our performance evaluation (Tables 2 and 3).

A.2 Description & Requirements

Our artifact represents a prototypical defect inference tool for Linux applications. As such, it must be executed within a Linux operating system environment. Our steps include the installing these tools within a virtual environment (e.g., within QEMU) running a specific Linux distribution as to avoid disruption of personal or organizational equipment.

A.2.1 Security, privacy, and ethical concerns

Our prototype requires no disabling of security mechanisms, does not interfere with the behavior of local programs, nor does it send any data to remote services. As such, there are no immediate privacy risks to users associated with the use of this tool.

At the same time, eBPF is not a mature software and is highly complex. During the preparation of this artifact, we identified multiple kernel versions where system integrity or availability could be affected due to bugs in the implementation of the userspace hooking of eBPF. Moreover, there is potential for sensitive data to be collected and stored *locally* within the report logs of our prototype. Finally, we cannot guarantee that no software supply chain attacks have occurred which expose the user to malicious code. To avoid potential system disruption and inadvertent collection of sensitive information, we suggest conducting all reproduction studies with our prototype within the proposed virtual environment. Further experimentation outside of this environment is strictly at the risk of the user.

A.2.2 How to access

Our stable artifact is made available under the Zenodo DOI:

https://doi.org/10.5281/zenodo.17979528

A "living" artifact is also made available on GitHub:

https://github.com/os-sanitizer/os-sanitizer

A.2.3 Hardware dependencies

The only requirement of evaluating our artifact is the use of an x86_64 host with at least 8 GB of RAM and the ability to launch x86_64 guests within a hypervisor. There may be slight differences on our evaluation hardware versus those used by the reviewers. Our reported figures are presented in terms of ratios, and we do not expect these values to fluctuate greatly.

A.2.4 Software dependencies

The most critical software dependency for the reproduction of our experiments is the SPEC CPU2017 benchmark. We

specify how this must be installed in Section A.3, but users must acquire a license for this software themselves.

Our software depends on many Fedora Linux packages, Rust programming language crates, and the Rust compiler. Provided that the virtual machine described in Section A.3 is correctly initialized with a network connection, all Rust dependencies will be automatically fetched during the build process. Furthermore, we provide a setup script which acquires all compilation dependencies within the artifact itself under `evaluation/setup_ae.sh`.

Users may utilize the hypervisor of their choice for conducting the experiments in a virtual environment, provided it is capable of running a typical Fedora Linux 42 installation.

A.2.5 Benchmarks

Beyond the SPEC CPU2017 benchmark described above, the microbenchmarks described in the paper may be found under the `examples/` directory in the artifact and the Browserbench "Speedometer 3.0" benchmark may be found at the URL: https://browserbench.org/Speedometer3.0/.

A.3 Set-up

As previously mentioned, users should first prepare a working hypervisor on their host system as a prerequisite. Then they may begin the installation steps below.

A.3.1 Installation

First, users should download the Fedora Linux 42 Workstation Edition ISO and the Fedora Linux 42 Server Edition ISO in their host environment. They should create two VMs, each with at least 4GB RAM, 30GB disk space, and a network connection, and on one install Fedora using the Workstation ISO using the typical steps and on the other the Server ISO.

With these installed, users may now download OS-SANITIZER on both machines with the following steps as root on each guest, replacing `$GIT_URL` with the GitHub URL provided above:

```
# dnf install git
# git clone $GIT_URL \
  --rev aad89c15c278e82200917314c0a045ea9fb7afe2
# evaluation/setup_ae.sh base
```

With these commands executed, the base installation is complete. Additional installation necessary for specific experiments is provided in the corresponding preparation step in Section A.4.2.

A.3.2 Basic Test

To ensure that OS-SANITIZER was installed correctly, reviewers can start the `os-sanitizer` utility with the following command:

```
# env RUST_LOG=info os-sanitizer --all
```

Within a few seconds, you should see alerts from local programs. The program will terminate within a few seconds of entering `CTRL-C`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): OS-SANITIZER induces low performance overhead during typical heavy workloads, though suffers when applied to frequently-invoked low-level userspace functions. This is demonstrated by experiments (E1) and (E2) and corresponds to Table 3 of the original paper.

(C2): OS-SANITIZER induces moderate performance overhead at the individual operation level and severe performance overhead for low-level userspace functions. This is demonstrated by experiment (E3) and corresponds to Table 2 of the original paper.

(C3): OS-SANITIZER induces negligible performance overhead during normal system usage. **We do not seek to reproduce this claim in this artifact evaluation** as this claim is qualitative and requires significant reviewer effort.

(C4): OS-SANITIZER identifies code regions likely to contain defects. **We do not seek to reproduce this claim in this artifact evaluation** as this claim is qualitative and requires significant reviewer effort. Reviewers who seek to replicate these findings may do so by installing the corresponding vulnerable versions of the softwares mentioned within the appropriate VM, e.g., by installing Docker version 26.1.3, or by viewing reports emitted by microbenchmarks provided under `examples`.

A.4.2 Experiments

(E1): [SPEC CPU2017] [30 human-minutes + 5400 compute-minutes for `-all` + 10GB disk] Establishes the performance impact of OS-SANITIZER during industry-standard heavy workloads that one might observe on servers and developer machines. This experiment takes place on the Server VM.
Preparation: Ensure that you have followed all installation instructions for the Server VM. Then, place the SPEC CPU2017 installer ISO file on the machine, then run

```
# evaluation/setup_ae.sh spec <spec.iso>
```

This script has been tested with `cpu2017-1.1.0.iso` (MD5: `f26e4f9404c31b82d40be2d71971d1c0`).
Execution: The scripts inside `evaluation/spec/` will

launch OS-SANITIZER with the correct configuration for that experiment. In total we have 18 different evaluation scripts which we recommend running in a VM of its own. For example, `eval_SPEC_intspeed_00.sh` should be copied to base directory where SPEC CPU2017 is installed and launched with `bash -x eval_SPEC_intspeed_00.sh`. Consequently, this will populate `cpu_intspeed_log_0_all_...` in base SPEC directory as well as results in child `result` directory.

Results: Files from 18 different VMs need to be arranged in the directory structure described in `evaluation/filtering/filter_SPEC.py`. Execution of `filter_SPEC.py` will populate filtered results into `evaluation/data/data_SPEC.json`. Additionally, Browserbench experiments needs to be performed to regenerate Table 3 from the paper as described in **E2**.

Due to time constraints, we recommend executing the "no passes (17)", "reference policy (16)", "all passes (00)", and two additional configurations of the reviewer's choice on three benchmarks of the reviewer's choice, instead of the ten benchmarks of the original paper. This reduces reviewer workload without excessively sacrificing rigor, as we have no mechanism by which to predict which experiments will be selected.

(E2): [Browserbench] [10 human-minutes + 3 compute-hours + 5GB disk] Establishes the performance impact of OS-SANITIZER on heavy browers-based workloads that one might observe on user-facing devices. This experiment takes place on the Workstation VM as it requires a *headful* environment.
Preparation: Ensure that you have followed all installation instructions for the Workstation VM. To install Browserbench specific dependencies, then execute:

```
# evaluation/setup_ae.sh browserbench
```

Execution: This experiment may be launched directly with the following commands.

```
# cd evaluation/browserbench
# bash -x eval_puppet_speedometer3.sh
# python3 ../filtering/filter_browserbench.py
    results/
```

Results: The `filter_browserbench.py` file aggregates results to the `evaluation/data/data_speedometer.json`, which is used in the shared output script (`output_SPEC_and_browser_print.py`). Since the paper uses a combined table for data from E1 and E2, both files in `evaluation/data` have to present.

(E3): [Microbenchmarks] [30 human-minutes + 1 compute-hour + 1GB disk] Establishes the performance impact of OS-SANITIZER on individual operations by means of benchmarking short sequences of operations repeatedly.
Preparation: Ensure that you have followed all installation instructions for the Server VM.
Execution: This experiment may be launched directly with the following commands.

```
# cd evaluation/microbenchmark
# bash eval_microbench.sh | \
    tee stdout_of_eval_microbench
# bash ../filtering/filter_microbench_data.sh
```

Results: Run the Python script `../output/output_microbenchmark_print.py` to generate the LaTeX code for Table 2 in the paper.

A.5 Notes on Reusability

OS-SANITIZER is merely a prototype to demonstrate the capacity of eBPF to be used to identify likely defective code regions. Alas, it is just that: a prototype. There are several usability gaps which we would like to address in the future.

For now, this prototype is best used via systemd units. A description of how to install OS-SANITIZER on a system for purposes other than evaluation, including installation as a systemd unit, is provided in the root README of the artifact's repository. Users intending to build their own passes are highly suggested to first review simpler passes like `system_absolute` pass under `os-sanitizer-ebpf`, observing how the information flows from eBPF to userspace via the components provided in the `os-sanitizer-common` crate, and finally is processed in userspace in the main loop of the `os-sanitizer` crate. More advanced passes will likely require the use of advanced helper functions, for which refer users to the corresponding Aya documentation and the examples provided by Michal R (AKA vadorovsky). We are happy to provide further support via the Discussions tab of the artifact repository, and will happily accept further contributions that make the development and usage of OS-SANITIZER easier 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Melting the Flesh of PHP's Memory Hardening

Yifan Wu
UC Riverside

Xiaochuan Yu
UC San Diego

Zhiyun Qian
UC Riverside

A Artifact Appendix

A.1 Abstract

This artifact reproduces the end-to-end exploits and targeted mitigations evaluated in our paper. It contains a uniform PHP-8.4.0 test environment with all four prototyped heap hardening protections enabled, five end-to-end exploit cases (three CVEs and two CTF challenges) that achieve remote command execution under this environment, and two prototype mitigations (Patch-hashtable and Patch-refcnt) that block these exploits. Each case is packaged as a one-click Docker setup and a Python exploit script, and each mitigation is provided as a one-click run script that rebuilds PHP with the patch applied and re-runs the exploit to confirm it is mitigated.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All exploits target a PHP interpreter running inside a self-contained Docker container, bound to localhost ports on the evaluator's host. No destructive operations are performed on the host. The exploit scripts only interact with the containerized target over HTTP and read back the output of `uname -a` and `date` as proof of remote command execution. The vulnerabilities exercised are existing public CVEs and CTF challenges, and no new vulnerabilities are introduced.

A.2.2 How to access

the archived copy of the artifact is available on Zenodo at https://zenodo.org/records/20401797. The more user-friendly Git repository is available at https://github.com/GhostFrankWu/PHP-security-research.

A.2.3 Hardware dependencies

A x86_64 machine with Docker support is sufficient for reproducing the exploit-success and mitigation claims. At least 8 GB of RAM and 20 GB of free disk space is recommended.

A.2.4 Software dependencies

Ubuntu 22.04 (or any Linux distribution with Docker support), Docker 24.0 or newer, Python 3.10+ with the `requests` and `tqdm` packages. All PHP toolchain and library dependencies are installed inside the provided Docker images.

A.2.5 Benchmarks

The overhead measurements for the two mitigations use Zend bench, Symfony Demo 2.2.3, and WordPress 6.2 as benchmark workloads. The benchmark drivers and target applications are included in the artifact and are invoked automatically by the provided scripts.

A.3 Set-up

A.3.1 Installation

Unpack the artifact archive and `cd` into the repository root. From the repository root, build the five exploit target images:

```
./build.sh  # Will build five Docker images as:
# ae-cve-2024-2961
# ae-cve-2022-31626
# ae-cve-2019-6977
# ae-ctf-case-a
# ae-ctf-case-b
```

Each build applies the two prototyped PHP defenses (`heap-isolation.patch` and `metadata-ro.patch`) on top of PHP-8.4.0 before compilation, so all builds reflect the same hardened configuration used in the paper. The mitigation evaluation scripts under `Patch-hashtable/` and `Patch-refcnt/` build their own images on demand.

A.3.2 Basic Test

Pick any one of the five built images, start it bound to its expected local port, and confirm the target PHP application responds:

```
docker run --rm -p 8082:80 ae-cve-2024-2961
# in another terminal:
curl -s http://127.0.0.1:8082/ | head
```

A successful basic test produces an HTTP response from the containerized PHP application without any visible error. The container can then be stopped before proceeding to the full evaluation workflow.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): All five exploits (CVE-2024-2961, CVE-2022-31626, CVE-2019-6977, CTF-Case-A, CTF-Case-B) achieve remote command execution against a PHP-8.4.0 interpreter with all four prototyped heap hardening protections enabled, demonstrated by the remote output of `uname -a` and `date`. This supports the central evaluation claim of the paper, summarized in Table 3 (column "Recon").

(C2): For the three CVE cases (CVE-2024-2961, CVE-2022-31626, CVE-2019-6977), the success rate and HTTP-request counts of the rebuilt exploits match the stability numbers reported in Table 4 of the paper.

(C3): Applying Patch-hashtable prevents the exploit from achieving command execution and from crashing the target. This patch covers the three CVE cases, consistent with the coverage discussed in §8.2.

(C4): Applying Patch-refcnt prevents the provided exploits in all five cases from achieving command execution and from crashing the target. The CTF-Case-A exploit path not covered by our prototypes has been independently addressed by two upstream patches (https://github.com/php/php-src/pull/16765 and https://github.com/php/php-src/pull/19287), as noted in §8.2.

A.4.2 Experiments

(E1): *End-to-end exploitation under heap hardening* [5 human-minutes + 20 compute-minutes + 8 GB disk]: each of the five exploits is run against its corresponding hardened target container and is expected to print the remote `uname` and `date` output. Supports (C1).
Preparation: Build the five target images as described in *Installation*.
Execution: For each case, start the target container bound to its expected port and run the exploit script from the repository root:

```
docker run --rm -p 8082:80 ae-cve-2024-2961
export URL=http://127.0.0.1:8082/
python3 CVE-2024-2961/exploit.py

docker run --rm -p 8083:80 ae-cve-2022-31626
export URL=http://127.0.0.1:8083/
python3 CVE-2022-31626/exploit.py

docker run --rm -p 8080:80 ae-cve-2019-6977
export URL=http://127.0.0.1:8080/
python3 CVE-2019-6977/exploit.py

docker run --rm -p 8084:80 ae-ctf-case-a
export URL=http://127.0.0.1:8084/index.php
python3 CTF-Case-A/exploit.py

docker run --rm -p 8081:80 ae-ctf-case-b
export URL=http://127.0.0.1:8081/dataform
python3 CTF-Case-B/exploit.py
```

Results: Each exploit prints the remote `uname` and `date` output on success. Successful runs of all five cases confirm (C1).

(E2): *Stability test on three CVE cases* [5 human-minutes + 2 compute-hours]: re-run each of the three CVE exploits 100 times and record the success rate and the number of HTTP requests sent. Supports (C2).
Preparation: Use the three CVE target images built in (E1).
Execution: For each of the three CVE cases, run its exploit script in a loop of 100 iterations against a freshly started container; record success/failure and the number of HTTP requests reported by the script.
Results: The observed success rate and the mean and worst-case request counts are expected to match Table 4 of the paper within normal variance. The CVE-2024-2961 and CVE-2022-31626 scripts first scan the default positive probe window and then a small negative fallback window. This covers hosts where the correct libc candidate sits a few pages before the default probe start. This confirms (C2).

(E3): *Patch-hashtable mitigation evaluation* [5 human-minutes + 10 compute-minutes]: apply Patch-hashtable on top of the hardened PHP build and re-run each of the three CVE exploits. Supports (C3).
Preparation: None beyond the artifact checkout; the script rebuilds PHP with the patch applied.
Execution: From the repository root:

```
export CLEANUP=1
Patch-hashtable/CVE-2024-2961/run.sh
Patch-hashtable/CVE-2022-31626/run.sh
Patch-hashtable/CVE-2019-6977/run.sh
```

Results: For each CVE, the exploit no longer produces the `uname` or `date` output and the PHP worker process does not crash. This confirms (C3).

(E4): *Patch-refcnt mitigation evaluation* [5 human-minutes + 40 compute-minutes]: apply Patch-refcnt on top of the hardened PHP build and re-run all five exploits. Supports (C4).
Preparation: None beyond the artifact checkout.
Execution: From the repository root:

```
export CLEANUP=1
Patch-refcnt/CVE-2024-2961/run.sh
Patch-refcnt/CVE-2022-31626/run.sh
Patch-refcnt/CVE-2019-6977/run.sh
```

`Patch-refcnt/CTF-Case-A/run.sh`
`Patch-refcnt/CTF-Case-B/run.sh`
Results: For each of the five cases, the provided exploit no longer produces the `uname` or `date` output and the PHP worker process does not crash. This confirms (C4).

A.5 Notes on Reusability

The artifact is structured so that each of the five exploit cases and each of the two mitigation prototypes is self-contained, i.e., a separate Dockerfile, a separate exploit script, and a separate mitigation run script. Adding a new exploit case requires only a new directory with a Dockerfile and a Python exploit script following the `URL` convention. Evaluating an additional mitigation on the same five cases requires only a new patch directory mirroring the layout of `Patch-hashtable/` or `Patch-refcnt/`, since the run scripts are parameterized by patch path and target case. The same harness can be reused to test other prototype defenses on top of the hardened PHP-8.4.0 baseline.

Beyond verifying that the exploits are mitigated, the artifact also supports measuring the runtime overhead of the two mitigations against Zend bench, Symfony Demo 2.2.3, and WordPress 6.2 using the scripts provided under `Patch-hashtable/` and `Patch-refcnt/`. The overhead measurements reported in the paper were collected on an Intel Core Ultra 9 285HX (24 cores) with 64 GB DDR5-4400 RAM and a 2 TB NVMe SSD, so reproducing the runtime overhead numbers benefits from a similar configuration. The reported full-run runtime overhead is approximately 0.17% for Patch-hashtable and approximately 8.86% for Patch-refcnt, with absolute timings dependent on hardware.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: RETROFIT: Continual Learning with Controlled Forgetting for Binary Security Detection and Analysis

Yiling He[1,⋆✉], Junchi Lei[2,⋆], Hongyu She[2],
Shuo Shao[2], Xinran Zheng[1], Yiping Liu[3], Zhan Qin[2], Lorenzo Cavallaro[1]
[1] University College London, London, United Kingdom
[2] Zhejiang University, Hangzhou, China
[3] Alibaba Group, Beijing, China

A Artifact Appendix

A.1 Abstract

Deep learning models for binary security degrade as threat landscapes shift and code representations evolve. Continual learning is a natural fit, but existing approaches either replay historical data or permit unconstrained updates, limiting applicability in data-sensitive settings. In this paper, we propose RETROFIT, which controls forgetting at each update without accessing historical data. It consolidates previously trained and newly fine-tuned models as teachers of legacy and emergent knowledge via parameter merging. Forgetting control is achieved via low-rank and sparse subspace constraints, with a confidence-guided arbitration dynamically aggregating knowledge from both teachers. On malware detection under temporal drift, RETROFIT improves retention (PTR metrics) by 20.2%-38.6% over continual learning baselines, surpassing the oracle upper bound on new data. On binary summarization across decompilation levels, it achieves over 2× the BLEU score of prior transfer-learning approaches on stripped binaries and outperforms all baselines in cross-representation generalization.

This artifact highlights two goals: (1) confirming that the implementation is available and functional, and can reproduce the main reported metrics and trends of RETROFIT presented in the paper; and (2) confirming that the complete experimental workflow, equipped with end-to-end scripts for both tasks covering training, parameter merging, and evaluation, is available and functional.

A.2 Description & Requirements

The artifact contains two independent implementation directories: `Malware_Detection/` and `Binary_Analysis/`, which should be run in their corresponding dependency environments.

⋆ Co-first authors. ✉ Corresponding author: heyilinge0@gmail.com

A.2.1 Security, privacy, and ethical concerns

The malware-detection workflow uses pre-extracted Android feature dictionaries and does not require evaluators to execute APKs. The binary-analysis workflow uses public datasets and pretrained models and does not execute unknown binary programs. The artifact does not collect personal information.

A.2.2 How to access

The artifact is permanently available through the Zenodo concept DOI:

https://doi.org/10.5281/zenodo.20339140.

The concept DOI represents all versions of the artifact. The README describes the repository layout, dependencies, data preparation, and experiment commands. Due to licensing and redistribution constraints, we provide instructions for obtaining the Transcendent dataset, the CAPYBARA dataset, and the CodeT5-base model through their official channels. To further facilitate reproducibility, we also provide the processed Transcendent features and the models used in our main results.

A.2.3 Hardware dependencies

The paper-level results were obtained on NVIDIA RTX A6000 GPU servers. `Malware_Detection/` can also run on CPU. `Binary_Analysis/` is more computationally demanding, so CUDA-capable NVIDIA GPUs are recommended for the full workflow.

For both tasks, the artifact provides lightweight evaluation or plotting paths using released intermediate artifacts, allowing the main reported figures to be obtained quickly on CPU without rerunning the complete training pipelines. For the `Malware_Detection/` lightweight reproduction path, the released processed inputs hosted on Hugging Face occupy approximately 15 GB, including both training and test data. The Figure 5 evaluation-only path requires approximately 3.8 GB of test inputs.

A.2.4 Software dependencies

The two components should be installed in separate environments using `Malware_Detection/requirements.txt` and `Binary_Analysis/requirements.txt`. The binary-analysis workflow uses the BinT5 Docker environment with Python 3.10.

The requirements files specify the Python-package dependencies required for evaluation and visualization; the binary-analysis workflow additionally relies on the BinT5/CodeT5 environment described in the README.

A.2.5 Benchmarks

The benchmarks cover malware detection under temporal drift and binary summarization across decompilation levels. For malware detection, we use the Transcendent dataset and MLP classifiers. For binary summarization, we follow the CAPYBARA setting and use a CodeT5-based backbone.

A.3 Set-up

This section gives the minimal setup steps for basic checks; full setup and reproduction commands are in the README.

A.3.1 Installation

For `Malware_Detection/`, install:

```
pip install -r \
    Malware_Detection/requirements.txt
```

For the Figure 5 evaluation-and-plotting workflow, download the released malware-detection models and dimension-reduced Transcendent inputs on Hugging Face: mlp-transcendent-cl. Place the released checkpoints under `Malware_Detection/model/` and the dimension-reduced input files under `Malware_Detection/input/`.

For `Binary_Analysis/`, use the BinT5-compatible environment in the README. A typical container setup starts with:

```
docker pull aalkaswan/bint5
docker run -i -t \
    --name retrofit_binary \
    --gpus all \
    -v $(pwd):/data \
    aalkaswan/bint5 /bin/bash
cd /data/
```

The binary-analysis checkpoints are released on Hugging Face: codet5-capybara-retrofit.

A.3.2 Basic Test

For `Malware_Detection/`, run:

```
python Malware_Detection/run_all.py check
```

This command performs a lightweight check of the training, evaluation, and drawing entry points used by the complete malware-detection workflow. It does not train models, read benchmark data, require released checkpoints, or require GPU execution. A successful run exits with status code 0 without Python tracebacks or error messages.

For `Binary_Analysis/`, run the following check inside the BinT5 environment:

```
python - <<'PY'
import sys
import torch, transformers, peft, safetensors
sys.path.insert(0, 'Binary_Analysis/Util')
import retrofit_config
print('RETROFIT binary basic test passed')
PY
```

The malware-detection check succeeds silently, while the binary-analysis check should print `RETROFIT binary basic test passed`.

A.4 Evaluation workflow

This section covers the RETROFIT results in Figure 5 and Figure 6.

A.4.1 Major Claims

(C1): In malware detection under temporal drift, RETROFIT adapts to newly arriving yearly data while mitigating forgetting on historical years, improving retention over continual-learning baselines while maintaining competitive performance on new yearly data. This claim corresponds to Figure 5 in the paper and the malware-detection portion of Table 1. The artifact supports this claim through the `Malware_Detection/` workflow, which covers preprocessing, continual-training, RETROFIT parameter merging, evaluation, and Figure 5 generation, including year-wise old/new F1, AUT, and PTR metrics.

(C2): In binary summarization across decompilation levels, RETROFIT improves continual adaptation and cross-representation generalization. This claim corresponds to Figure 6 in the paper. The artifact supports this claim through the RETROFIT reproduction workflow in `Binary_Analysis/`, using released fine-tuned and RETROFIT-merged BinT5 checkpoints for direct evaluation and providing scripts to reproduce the RETROFIT evaluation and Figure 6 results.

A.4.2 Experiments

(E1): Malware detection under temporal drift (C1) [about 5–10 human-minutes + about 3.5 GPU compute-hours for the complete RETROFIT workflow; about 21 minutes for the Figure 5 evaluation-and-plotting workflow with released artifacts].
How to: We provide two reproduction paths. The first path runs the complete RETROFIT workflow, including yearly training, RETROFIT parameter merging, evaluation, and figure generation. The second path is the Figure 5 evaluation-and-plotting workflow, which uses released models and dimension-reduced inputs as a lightweight reproduction check. Its evaluation output also contains the year-wise old/new F1 values used for the malware-detection part of Table 1.
Preparation: Install dependencies following the README. For the complete RETROFIT workflow, prepare the processed Transcendent feature files and yearly train/test splits. For the Figure 5 evaluation-and-plotting workflow, download the released checkpoints and the required dimension-reduced test inputs.
Execution: To run the complete RETROFIT workflow, execute:

```
python Malware_Detection/run_all.py all
```

This command trains the 2014 base model, performs yearly updates from 2015 to 2018, merges old and newly adapted models with RETROFIT, evaluates yearly F1/AUT/PTR metrics, and draws Figure 5.
To run the Figure 5 evaluation-and-plotting workflow directly from the released models, execute:

```
python Malware_Detection/run_fig5.py
```

This command runs `eval_fig5.py` to compute year-wise old/new F1 scores, AUT, and PTR from the released checkpoints and reduced inputs, and writes them to `fig5_results.json`. The old/new F1 entries in this JSON provide the malware-detection values for Table 1, while `draw_fig5.py` uses the same evaluation output to render Figure 5.
Results: The complete workflow produces trained yearly models, evaluation summaries, and Figure 5. The Figure 5 evaluation-and-plotting workflow produces `fig5_results.json`, from which the malware-detection old/new F1 values in Table 1 can be obtained, and generates the Figure 5 visualization. Due to differences in hardware, software versions, random seeds, and numerical non-determinism, the reproduced values may differ slightly from the exact values in Figure 5, but the overall trend should remain unchanged: RETROFIT retains stronger historical performance while maintaining competitive adaptation to newly arriving yearly data.

(E2): Binary summarization across decompilation levels (C2) [about 10–20 human-minutes + about 70 GPU compute-hours for the complete RETROFIT training workflow; about 30 seconds for the Figure 6 plotting workflow from released output files].
How to: We provide two lightweight reproduction paths that can be run on CPU. The first path runs the binary-summarization evaluation workflow to generate summaries and metrics. The second path is the Figure 6 plotting workflow, which computes the plotted values directly from existing `Binary_Analysis/Output/` files.
Preparation: Configure the CodeT5-based environment and CAPYBARA data following the README.
Execution: Run the test interface:

```
CODET5_SH_DIR=/path/to/CodeT5/CodeT5/sh \
    bash Binary_Analysis/CodeT5_test.sh
```

To compute additional metrics, run:

```
python Binary_Analysis/Eval_Metrics.py \
    --base_dir Binary_Analysis/Output
```

To draw Figure 6 directly from the evaluation outputs, run:

```
python Binary_Analysis/draw_fig6.py
```

Results: The evaluation workflow generates summaries and computes BLEU and auxiliary generation metrics. The Figure 6 plotting workflow reads `Binary_Analysis/Output/`, computes the plotted values, and outputs the Figure 6 content in about 30 seconds. For consistency with the original BinT5 comparison, the Base/stripped cell in Figure 6 follows the BinT5 paper's reported result (7.19).

For workflows involving retraining or fresh inference, exact values may differ slightly from the paper because of hardware differences, random seeds, and non-determinism; the expected result is to reproduce the same overall trends.

A.5 Notes on Reusability

The two implementation directories can be reused independently: `Malware_Detection/` for temporally organized malware features, and `Binary_Analysis/` for CodeT5-based summarization tasks prepared in the upstream CodeT5 format.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: SafeFFI: Efficient Sanitization at the Boundary Between Safe and Unsafe Code in Rust and Mixed-Language Applications

Oliver Braunsdorf[1], Tim Lange[1], Konrad Hohentanner[2], Julian Horsch[2], and Johannes Kinder[1]

[1]Ludwig-Maximilians-Universität München, Germany
[2]Fraunhofer AISEC, Germany

A Artifact Appendix

A.1 Abstract

This document describes the artifact accompanying the SafeFFI paper. This includes our modifications to rustc and LLVM, test sets, benchmarks and the corresponding scripts to build and execute them with SafeFFI, as well as our raw evaluation data. A more detailed listing of files can be found in `ArtifactContentsOverview.md`.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact uses a VSCode devcontainer to build and test our custom rustc, which requires "seccomp=unconfined" privileges for running gdb with ASLR disabled inside the container. Additionally, 3rd party software is downloaded from Rust's package registry crates-io for testing SafeFFI. This 3rd party code and accompanied build scripts are executed during testing and benchmarking. While they could potentially contain malicious code that would be executed inside the unconfined container, it is unlikely because the downloaded crates are popular within the Rust community. However, users of the artifact should be aware of these security implications and might want to run the artifact inside a virtual machine for better isolation.

A.2.2 How to access

The artifact files are available on Zenodo: https://doi.org/10.5281/zenodo.17976648

A.2.3 Hardware dependencies

The artifact builds the Rust[1] compiler and LLVM[2] and thus requires a machine with sufficient RAM and disk space. We recommend at least 20 cores, 32 GB of RAM and 100 GB of free disk space. The artifact is supported to work under Linux on x86-64 and AARCH64 architectures. Because our experiments are run and compared on two different architectures, reproduction requires a machine with an x86-64 processor and a machine with an AARCH64 processor.

A.2.4 Software dependencies

All scripts are executed within docker containers and VSCode devcontainers[3]. Thus, the artifact only requires a build host with a modern Linux distribution that has docker and docker-compose installed (make sure the user on the build host is part of the docker-group on Linux), as well as Visual Studio Code (VSCode) on the evaluator's PC. For the build server, we used Ubuntu 24.04 on x86-64 and Fedora Linux Asahi Remix 39 on AARCH64 (MacStudioM2). We recommend the following commands to install the necessary dependencies for docker on Ubuntu:

```
sudo apt update
sudo apt install -y docker.io docker-compose
sudo usermod -aG docker $USER
newgrp docker
```

We recommend building and executing to be done on a remote build server with the above-mentioned hardware and software dependencies, connecting from the evaluator's PC via VSCode's remote development features.

[1] https://rustc-dev-guide.rust-lang.org/building/prerequisites.html?highlight=RAM#hardware
[2] https://llvm.org/docs/GettingStarted.html#hardware
[3] https://code.visualstudio.com/docs/devcontainers/containers

A.2.5 Benchmarks

No external benchmarks. All benchmarks used for the evaluation are contained in the artifact.

A.3 Set-up

To build the prototype, first, LLVM needs to be built, then rustc. Afterwards, the prototype can be tested. Note that detailed instructions and information about build flags can be found in the `README.md` in the artifact. However, for one-off building for artifact evaluation purposes, we provide basic commands for installation and testing below.

A.3.1 Installation

Prepare the host system (VM) Set entropy for ASLR to 28 bits, e.g. by using the following command as root on your host system (or inside a VM if you use one to isolate the artifact):

```
sudo sysctl -w vm.mmap_rnd_bits=28
```

> **Note**
>
> Please make sure to not skip setting the `vm.mmap_rnd_bits`. Due to a bug in the older versions of the vanilla LLVM sanitizers (https://github.com/google/sanitizers/issues/1614), this could otherwise lead to a false positive sanitizer detection which might affect reproducibility of our results.

Download and Patch LLVM and Rustc For a better overview of the changes, we provide our modifications to LLVM and rustc as git patches. To prepare the source code for building, run the following commands in the root of the artifact directory:

```
# go to root of directory
cd SafeFFI

# shallow clone and patch of llvm
git clone --depth 1 \
  https://github.com/rust-lang/llvm-project.git
cd llvm-project
git fetch --depth 1 \
  origin c3a26cbf6e73f2c5f8d03cee1f151d90a266ef3c
git checkout c3a26cbf6e73f2c5f8d03cee1f151d90a266ef3c
git apply --whitespace=nowarn ../llvm-patch.diff
cd ..

# shallow clone and patch of rust
git clone --depth 1 https://github.com/rust-lang/rust
cd rust
git fetch --depth 1 \
  origin 61edfd591cedff66fca639c02f66984f6271e5a6
git checkout 61edfd591cedff66fca639c02f66984f6271e5a6
git apply --whitespace=nowarn ../rust-patch.diff
cd ..
```

Build LLVM. Run the following commands in the root of the artifact directory:

```
cd SafeFFI/llvm-project
./local-build-with-docker.sh
```

Build rustc. For building rustc we provide a VSCode devcontainer.

- Open the directory (`SafeFFI/rust/`) in VSCode (via `File -> Open Folder`) and then press `F1 -> Dev Containers: Reopen in Container` to open the rust directory inside the VSCode devcontainer.
- Note: if VSCode seems to hang while building the devcontainer, pressing `F1 -> Reload Window` might help.
- When the container successfully spawned, open a new terminal inside VSCode. This should point you to the working directory `/workspaces/SafeFFI/SafeFFI` with the whole artifact mounted as bind-mount.
- For building our custom rustc **for the first time**, do:

  ```
  cd rust/SafeFFI-Docker/
  ./initial_build.sh
  touch /tmp/dealloc_functions.txt
  ```

- This produces our custom stage1 rust compiler that can be invoked with `cargo +stage1 <build command>`

> **Note**
>
> From here every other step for testing and evaluating happens inside the VSCode devcontainer.

Install python requirements Execute the following commands to install dependencies for running the artifact's python scripts:

```
python3 -m venv /home/vscode/safeffi-venv
source /home/vscode/safeffi-venv/bin/activate
pip install matplotlib pandas scipy
```

A.3.2 Basic Test

The `safeffi-tests` directory is also mounted in the devcontainer container at '/safeffi-tests'. You can use the tests to verify the SafeFFI Rust compiler.

```
cd /safeffi-tests/tests/rust/simple_sanitizer_tests
./safeffi_build.sh
```

The script runs two simple tests for out-of-bounds and use-after-free memory safety violations with SafeFFI on AddressSanitizer (ASan). They are succesful if the output contains `ERROR: AddressSanitizer: stack-buffer-overflow` and `ERROR: AddressSanitizer: heap-use-after-free`

Double-check `vm.mmap_rnd_bits` Running the following command in the docker container should print the `vm.mmap_rnd_bits` that you set earlier.

```
sudo sysctl vm.mmap_rnd_bits
```

A.3.3 Minimal Working Example

Below you can see the minimal working example of a command to build and run a program with SafeFFI and ASan. This command works in cargo-based build systems for Rust and mixed-language (including C/C++) code.

```
CC="/llvm-project/docker-install-dir/bin/clang" \
CFLAGS="-fsanitize=address" \
RUSTFLAGS="-Zsafeffi \
-Zsafeffi_temporal_mode=reinsert_checks_after_calls:/
    tmp/dealloc_functions.txt \
-Cllvm-args=-safeffi-nofree-out=/tmp/
    dealloc_functions.txt \
-Zsanitizer=address \
-Clinker=/llvm-project/docker-install-dir/bin/clang \
-Clink-arg=-fuse-ld=lld" \
cargo +stage1 run \
--target=aarch64-unknown-linux-gnu
```

We explain the flags in more detail here. The first two flags are for the C/C++ compiler.

- `CC="/llvm-project/docker-install-dir/bin/clang"`: specify the clang compiler compatible with our LLVM toolchain.
- `CFLAGS="-fsanitize=[address|hwaddress]"`: enable the sanitizer for C/C++ code.

The remaining flags inside of `RUSTFLAGS` are consumed by the Rust compiler.

- `Zsafeffi`: activate SafeFFI for Rust source code (this implicitly adds `-hwasan_safeffi_propagation_elision` to the rustc's invokation of LLVM's sanitizers to activate the SafeFFI-Lib)
- `Zsafeffi_temporal_mode`: activate(`reinsert_checks_after_calls`) or deactivate (`reinsert_none`) SafeFFI's additional checks for Free-During-Scope vulnerabilities. For reinserting checks you need to specify a file that should exist and be empty: `reinsert_checks_after_calls=</path/to/noFreefunctions>`
  This is where our on-the-fly callgraph analysis writes the functions that are classified using our NoFree SCC Pass as descibed in §6.4 of the paper.
- `-Cllvm-args=-safeffi-nofree-out` has to point the the same file if the second option of the previous flag is used.
- `-Zsanitizer=[address|hwaddress]` : selects the underlying sanitizer for Rust (this has to match the sanitizer for C++). If Hardware-assisted AddressSanitizer (HWASan) is selected an additional `-Ctarget-feature=+tagged-globals` is necessary.
  `-Clinker=/llvm-project/docker-install-dir/bin/clang` :
  specify the clang linker compatible with our LLVM toolchain.
- `-Clink-arg=-fuse-ld=lld` : specify the use of the lld linker that is compatible with our LLVM toolchain.
- `cargo +stage1 run`: invokes SafeFFI's custom stage1 rust compiler to build and run the program.
- `--target=[x86_64|aarch64]-unknown-linux-gnu` : necessary for Rust to sanitize the Rust application but not the `build.rs` scripts.

We included an example of a mixed-language application that can be built with this command in the directory `/safeffi-tests/tests/ffi/minimal_working_example`. It contains an out-of-bounds vulnerability that is detected by SafeFFI's inserted checks when running this example with the above command.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SafeFFI correctly detects all memory safety violations that the underlying sanitizer detects in a set of known real-world memory safety vulnerabilities. This is proven by experiment E1 which is described in Section 7.2 of the paper whose results are reported in Table 1.

(C2): SafeFFI effectively reduces the number of sanitizer checks compared to the instrumentation with vanilla ASan. This is proven by experiment E2 which is described in Section 7.3 "Elided Checks" of the paper whose results are illustrated in Figure 5.

(C3): SafeFFI reduces the run-time overhead of ASan and HWASan. This is proven by experiment E2 which is described in Section 7.3 Paragraphs "Run-time Performance" and "ARM and HWASan" of the paper whose results are illustrated in Figures 6 and 7.

(C4): SafeFFI achieves superior compile-time overhead compared to state-of-the-art approaches. This is proven by experiment E2 which is described in Section 7.5 of the paper whose results are illustrated in Figure 8.

(C5): SafeFFI works robustly in Mixed-Language Application (MLA) scenarios where Rust and C/C++ interoperate via FFI and does not produce false positives nor false negatives. This is proven by experiment E3 which is described in Section 7.4 of the paper.

A.4.2 Experiments

The experiments described below describe how to reproduce the major claims about SafeFFI made in the paper. We do not include reproduction of test results from related work (ERASan and RustSan) as these are outside the scope of our artifact.

To reproduce results for SafeFFI on HWASan, the experiments need to be executed on an AARCH64 machine since HWASan is only supported for that architecture. For ASan, the experiments can be executed on both x86-64 and AARCH64 machines. Table 1 summarizes the exact environment we used for our experiments.

Please don't execute the experiments in parallel on the same host as this could lead to resource contention and unreliable results.

(E1): Correctness [10 human-minutes + 60 compute-minutes]: Run a set of known real-world memory safety vulnerability PoCs compiled with SafeFFI and check that all memory safety violations are detected.

Preparation: Ensure that the SafeFFI Rust compiler is built inside the VSCode devcontainer as described in Section A.3.

Execution: Inside the VSCode devcontainer, run:

```
cd /safeffi-tests/tests/rust
./test-all-pocs-with-safeffi.py
```

Results: The results are printed to the console in a summarizing table that shows the detection results for each PoC and each configuration of sanitizer configuration. The result classification for vanilla sanitizers is `DETECTED` or `FAILED_TO_DETECT`. For SafeFFI, the additional classifications `cast-check` and `temporal-check` indicate SafeFFI-specific detections. The results should match Table 1 in the paper. As indicated in the paper: "Some vulnerabilities in the dataset are non-deterministic which makes it necessary to run each compiled test binary multiple times to observe the vulnerability reliably, especially for HWASan because its probabilistic detection mechanism adds further non-determinism on top, even for deterministic vulnerabilities. If at least one run triggered the vulnerability, we classified it as detected." Thus, the script executes each compiled PoC binary 100 times and the result output shows the number of executions per PoC in which the sanitizer failed to detect the vulnerability. For HWASan-based configurations, we have seen failure rates of up to ∼10 times out of 100 executions.

(E2): Check Elision and Performance Benchmarks [30 human-minutes + 10 compute-hours]: Run a set of benchmarks to measure the number of sanitizer checks elided by SafeFFI, as well as compile-time overhead and runtime overhead compared to vanilla ASan/HWASan and a baseline without instrumentation.

Preparation: Ensure that the SafeFFI Rust compiler is built inside the VSCode devcontainer as described in Section A.3. Initialize the frozen crates-io index as git repository:

```
cd /playground
rm -rf \
  crates.io-index-archive-snapshot-2020-11-20/
git clone \
  https://github.com/SafeFFI/safeffi-crates-io-
    index.git \
  crates.io-index-archive-snapshot-2020-11-20
```

To check if the benchmarks can be correctly downloaded and run in your setup, you can first execute:

```
cd /playground/rust-benchmarking-scripts
./run.py --smoke
```

This executes a small subset of the benchmarks to verify that everything is working correctly(ca. 20-30 minutes on X86, ca. 40-60 minutes on AARCH64). You should see the script terminating without errors and produce the same CSV files as listed below.

Execution: Inside the VSCode devcontainer, run:

```
cd /playground/rust-benchmarking-scripts
./run.py
```

> **Note**
>
> Some of the benchmarks are very long-running, especially if instrumented with non-optimized sanitizers. We set a timeout for each individual benchmark execution to 120 minutes which should be sufficient to finish all benchmarks. You might want to adjust the threshold variable `BENCH_TIMEOUT` in `run.py:22` to make the benchmarking process either faster or more robust against timeouts on your evaluation hardware. During Usenix Artifact Evaluation, four benchmarks (Chrono, Adler, Base64, and Image) failed in the vanilla ASan and SafeFFI-ASan versions on some systems. Only those four benchmarks are using the Criterion crate as a benchmarking backend. The failure pattern indicates that the common reason is a bug in Criterion (we used it's older version 0.3.2) which triggers ASan[a].
>
> [a] Criterion's custom `black_box()` has the same issue as: https://github.com/rust-lang/rust/pull/104110/

Results: The results are stored as CSV files in `/playground/rust-benchmarking-scripts/tmp` and contain the absolute numbers for the measurements that resemble the results reported in the paper.

- `check_elision_stats.csv` (Claim C2)
- `runtime_stats.csv` (Claim C3)
- `compilation_stats.csv` (Claim C4)

Table 1: The exact environment we used for our experiments.

Architecture	CPU	RAM	Host-OS	Container-OS
x86-64	AMD EPYC 9645 (384 cores)	1.5 TB	Ubuntu 24.04	Ubuntu 20.04
AARCH64	Apple M2 Ultra (24 cores)	192 GB	Asahi Linux 6.12.0	Ubuntu 20.04

(E3): Robustness in MLA Scenarios [10 human-minutes + 10 compute-minutes]: Run a systematically created set of vulnerable and benign test cases of mixed-language applications with SafeFFI and check that the test cases are classified correctly. These tests require HWASan and therefore need to be run on an AARCH64 machine.
Preparation: Ensure that the SafeFFI Rust compiler is built inside the VSCode devcontainer as described in Section A.3.
Execution: Inside the VSCode devcontainer, run:

```
cd /playground/safeffi-systematic-tests/
./generation.py
```

Results: The results are printed to the console in a summarizing table that shows the detection results for each test case. The output is in Markdown table format. The test is successful if all test cases contain the string *"correctly"* in the column *Error Detected*. Log files for each individual test case are stored in the `log/` subdirectory.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Enhanced Private Set Union from Secret-shared Private Membership Test

Meng Hao[1], Guodong Wang[2], Xinpeng Yang[3], Pengzhi Xing[2], Hanxiao Chen[2,✉], Tianwei Zhang[3], Haiyang Xue[1], Guomin Yang[1], Hongwei Li[2,✉], Robert H. Deng[1]
[1]Singapore Management University
[2]University of Electronic Science and Technology of China
[3]Nanyang Technological University

A Artifact Appendix

A.1 Abstract

In this artifact appendix, we provide a self-contained roadmap for evaluating the artifact associated with our paper. We describe the required hardware, software, and network-emulation configuration, and explain how evaluators can build and run the artifact either natively on a Linux machine or inside the provided Docker environment. All implementations are written in C++ and include scripts for building dependencies, running basic functionality tests, and reproducing the main timing and communication results reported in the paper. Full-scale experiments may require substantial CPU time, memory, and privileged Linux network namespace support, the goal of this appendix is to make the artifact straightforward to inspect, build, execute, and compare with the paper results.

A.2 Description & Requirements

The artifact contains the implementations of our two ePSU variants: ePSU_fast and ePSU_low, together with the implementations of their main building blocks, including ssPMT, psPMT, and RPMT. Evaluators can use the artifact to reproduce the end-to-end ePSU results shown in Figure 24, Table 2, and Table 3 of our paper. These results report the running time and communication cost of the full ePSU protocol under different input sizes and network settings. Evaluators can also use the artifact to reproduce the component-level results shown in Figure 21 and Figure 22.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The artifact of our paper is publicly available on Zenodo at https://zenodo.org/records/20411092, with the version-specific DOI 10.5281/zenodo.20411092. This Zenodo record provides a stable archived version of the artifact associated with the camera-ready paper. Evaluators can download the artifact package from this record and use it to reproduce the experimental results reported in our paper.

The artifact root directory contains the README, Dockerfile, and two main implementation directories: ePSU_fast and ePSU_low. The ePSU_fast directory contains the fast ePSU implementation and the benchmark for ePSU. The ePSU_low directory contains the low-communication ePSU implementation and the related benchmark. Evaluators can follow the README to build the artifact, run the basic functionality tests, and reproduce the results for Figure 21, Figure 22, Figure 24, Table 2, and Table 3.

A.2.3 Hardware dependencies

The implementations are written in C++ and run on standard x86-64 Linux machines. The experimental results reported in the paper were obtained on a server equipped with two Intel Xeon Platinum 8368Q processors and 1 TB of RAM. For fair comparison, all baseline protocols were re-run under the same experimental setup, and both our protocols and the baselines were executed using a single thread.

Evaluators do not need access to this exact server to check the functionality of the artifact. However, full reproduction of the absolute running time reported in the paper may depend on using a machine with comparable CPU and memory resources. On different machines, evaluators should mainly compare the communication cost and the overall performance trends.

A.2.4 Software dependencies

The artifact is intended to run on Linux. We provide a Dockerfile based on Ubuntu 24.04, which installs the required system packages, including CMake, a C++ compiler, Git, Python 3, OpenMP, OpenSSL, GMP, NASM, iproute2, and other build tools. We recommend evaluators use the Docker environment to avoid manually configuring these dependencies.

The artifact also relies on third-party cryptographic libraries, including volepsi and secure-join. These libraries are downloaded and built automatically by the provided setup scripts. The benchmark scripts that emulate LAN and WAN settings require Linux network namespaces and the tc command from iproute2. Therefore, full benchmark reproduction should be run with sudo privileges or inside a privileged Docker container. No proprietary software, external dataset, or machine-learning model is required.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We recommend that evaluators build the artifact using the provided Dockerfile, since it installs the required system packages and compiles both ePSU implementations in a clean Linux environment. The artifact can also be built natively on Linux by running the setup scripts in the two implementation directories.

From the artifact root directory, evaluators can build and start the Docker container as follows:

1. Build the Docker image: `docker build -t epsu-sspmt .`
2. Start the container: `docker run -rm -it -privileged epsu-sspmt bash`

The `-privileged` option is required for the full benchmark scripts, which emulate LAN and WAN network settings using Linux network namespaces and `tc netem`.

Evaluators who prefer native installation can build the two implementations separately as follows:

1. Install the required system packages: `sudo apt-get update && sudo apt-get install -y -no-install-recommends ca-certificates openssl build-essential cmake git libtool autoconf automake pkg-config iproute2 python3 sudo nasm libssl-dev libgmp-dev wget libfmt-dev && sudo update-ca-certificates`
2. Build the fast implementation: `cd ePSU_fast && bash setup.sh`
3. Build the low-communication implementation: `cd ../ePSU_low && bash setup.sh`

A.3.2 Basic Test

After building the artifact, evaluators can run a small two-party ePSU execution to check that the required software components are built and functioning correctly.

For the fast implementation, run:

1. `cd ePSU_fast`
2. `bash ePSU_bench.sh 14`

For the low-communication implementation, run:

1. `cd ../ePSU_low`
2. `bash ePSU_bench.sh 14`

A successful run should end with:
`[SUCCESS] Finished.`
The output should also contain timing and communication information for the executed protocol. These numbers are only used to check functionality and are not expected to match the full paper results exactly.

A.4 Evaluation workflow

A.4.1 Major Claims

To reproduce the performance results shown in our paper, evaluators need to build the artifact and run the benchmark scripts provided in the two implementation directories, ePSU_fast and ePSU_low. The scripts execute two protocol parties on a single machine and record running time and communication cost under different input sizes and emulated network settings.

(C1): The end-to-end performance of our two ePSU variants can be reproduced by running the ePSU benchmark scripts in ePSU_fast and ePSU_low. The reproduced results correspond to Figure 24, Table 2, and Table 3 in the paper.

(C2): The performance of the PMT building blocks used in our ePSU constructions can be reproduced by running the ssPMT benchmark in ePSU_fast and the psPMT benchmark in ePSU_low. The reproduced results correspond to Figure 21 in the paper.

(C3): The performance of the RPMT building block can be reproduced by the artifact. This claim is evaluated by running the RPMT benchmark scripts in both ePSU_fast and ePSU_low. The reproduced results correspond to Figure 22 in the paper.

A.4.2 Experiments

(E1): End-to-end ePSU performance [20 human-minutes + several compute-hours]: Reproduce the end-to-end performance results of `ePSU_fast` and `ePSU_low` which corresponds to Figure 24, Table 2, and Table 3 in the paper.

Preparation: Use a Linux machine or a privileged Docker container, since the benchmark scripts use Linux network namespaces and `tc netem` to emulate LAN and WAN settings. Build both `ePSU_fast` and `ePSU_low` before running the experiment.
Execution: Run the following commands:

1. `cd ePSU_fast`
2. `sudo bash ePSU_batch_bench.sh`
3. `cd ../ePSU_low`
4. `sudo bash ePSU_batch_bench.sh`

Results: The scripts generate timestamped log directories such as `logs_ePSU_YYYYMMDD_HHMMSS`. The reproduced numbers should be compared with Figure 24, Table 2, and Table 3.

(E2): ssPMT/psPMT component performance [15 human-minutes + several compute-hours]: Reproduce the performance results of the PMT building blocks which corresponds to Figure 21 in the paper.
Preparation: Build both implementations. The fast implementation evaluates `ssPMT`, while the low-communication implementation evaluates `psPMT`.
Execution: Run the following commands:

1. `cd ePSU_fast`
2. `sudo bash ssPMT_batch_bench.sh`
3. `cd ../ePSU_low`
4. `sudo bash psPMT_batch_bench.sh`

Results: The scripts generate log directories report the running time and communication cost of the PMT components under different input sizes and network settings. The results should be similar to the trends reported in Figure 21.

(E3): RPMT component performance [15 human-minutes + several compute-hours]: Reproduce the performance results of the RPMT building block which corresponds to Figure 22 in the paper.
Preparation: Build both implementations.
Execution: Run the following commands:

1. `cd ePSU_fast`
2. `sudo bash RPMT_batch_bench.sh`
3. `cd ../ePSU_low`
4. `sudo bash RPMT_batch_bench.sh`

Results: The scripts generate logs report running time and communication cost for RPMT. The reproduced results should be compared with Figure 22. Absolute running time may vary across machines, while communication cost and scalability trends should remain consistent.

A.5 Notes on Reusability

The artifact is organized into two independent implementations, `ePSU_fast` and `ePSU_low`, so future users can study, modify, or benchmark each variant separately. The benchmark scripts can also be adapted to evaluate different input sizes or network settings by changing the set-size list and the `tc netem` parameters in the scripts.

The component benchmarks for `ssPMT`, `psPMT`, and `RPMT` may be useful beyond reproducing the paper results. Researchers can use these components to compare alternative building blocks or to evaluate new protocol optimizations under the same LAN and WAN emulation framework. The artifact does not require external datasets, so new experiments can be created by adjusting the protocol parameters and rerunning the provided scripts.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: SING: Improving the Efficiency of MPC Protocol Assignment using Graph Neural Networks

Jannis Blüml[1] Moritz Huppert[1] Nora Khayata[2] Joachim Schmidt[2] Thomas Schneider[2]
Technical University of Darmstadt, Darmstadt, Germany
[1]{firstname}.{lastname}@tu-darmstadt.de
[2]{khayata,joachim.schmidt,schneider}@encrypto.cs.tu-darmstadt.de

A Artifact Appendix

A.1 Abstract

The artifact consists of the codebase and datasets accompanying SING, a machine-learning-based framework for hybrid MPC share assignment. Specifically, we provide three archives: (i) the complete source directory tree of our codebase, including the pretrained models, (ii) the full unprocessed dataset of MPC circuits in C source code format, and (iii) the fully processed dataset of MPC circuits. The codebase includes a detailed `README.md` with installation and reproduction instructions.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

We release a permanently accessible version of our artifact on Zenodo[1]. This release comprises all code used as well as the complete dataset used to train our models. Additionally, we release our code on GitHub[2]. During the evaluation period, we will continue to update the GitHub repository as needed. Once the evaluation phase has concluded, we will (permanently) incorporate all changes into the Zenodo repository.

A.2.3 Hardware dependencies

Evaluating our pretrained share-assignment and cost-prediction models on reasonably small circuits does not require specialized hardware. Model inference runs on the CPU by default, although larger circuits require more RAM. We recommend at least 4 GB of free RAM for evaluating our benchmark circuits. Training the models from scratch, rather than using the provided pretrained models for inference, requires hardware acceleration. For training on the full dataset, we recommend a GPU with at least 48 GB of VRAM.

A.2.4 Software dependencies

We recommend running all experiments on Debian 13 or Ubuntu 26.04. While the model training and inference pipelines of SING are OS-agnostic, the external dependencies ABY [4] and Silph [3] are supported only on Linux. We note that for training and evaluating our share assignment and cost prediction models based on the existing (full) dataset, these third-party dependencies are not required. However, generating new data and benchmarking requires ABY to benchmark circuit runtimes and Silph to generate optimal assignments. Furthermore, benchmarking MPC runtime and communication using our benchmark harness requires root privileges in order to simulate different network environments.

SING Training and evaluation of our share assignment and cost prediction models requires a Python $\geq$ 3.13 installation. These packages are expected to be installed:

```
torch          torchvision
torchaudio     torcheval
torchmetrics   torch_geometric
tqdm           line_profiler
black          tabulate
wandb          pandas
plotly         kaleido
dash           scipy
```

Silph The following Debian packages are required:

```
curl           cmake
coinor-cbc     coinor-libcbc-dev
cvc4           git
libgmp-dev     libboost-all-dev
libssl-dev     time
zsh            build-essential
python3        python3-pip
```

[1] https://doi.org/10.5281/zenodo.20347718
[2] https://github.com/encryptogroup/SING

ABY The following Debian packages are required:

```
build-essential   cmake
libgmp-dev        libssl-dev
doxygen           libboost-all-dev
```

A.2.5 Benchmarks

All 9 benchmark circuits used in the paper are included in the full dataset available in our Zenodo repository (`dataset-full.tar.gz`). We list the circuit files and hashes used in the `benchmark-results/circuits.csv` file.

A.3 Set-up

First, download the (up to date) code archive from GitHub (`sing-artifact.tar.gz`) and the processed full dataset from Zenodo (`dataset-full.tar.gz`). As mentioned in Section A.2.2, we will update the GitHub repository with patches and fixes throughout the evaluation process. While the code repository (GitHub) includes a small excerpt of the dataset (`dataset-excerpt.tar.gz`), reproducing model training from scratch and our benchmarks requires the full dataset, which is hosted only on Zenodo due to its size.

A.3.1 Installation

SING We recommend setting up an isolated virtual environment and installing the packages listed in section A.2.4 in this environment. Run the following commands in the root of the code repository:

```
python -m venv venv
source venv/bin/activate

pip install torch torchvision torchaudio

pip install -r requirements.txt
```

The installation of PyTorch [1] may differ depending on your hardware. We refer to the official documentation[3] for further instructions.

ABY We have prepared a `Dockerfile` in the `ABY-vendor` subdirectory to provide a reproducible build process in an isolated environment.

```
docker build -t aby .
```

Alternatively, the build steps specified in the `Dockerfile` can be run on the host instead. Run the following commands in the `ABY-vendor` subdirectory of the code repository:

```
apt-get install \
build-essential \
cmake \
libgmp-dev \
libssl-dev \
doxygen \
libboost-all-dev

mkdir build
cd build
cmake .. -DCMAKE_BUILD_TYPE=Release \
-DABY_BUILD_EXE=On && make
```

Depending on the CMake version, it may be necessary to add the following flag to the `cmake` invocation: `-DCMAKE_POLICY_VERSION_MINIMUM=3.5`.

Silph We have prepared a `Dockerfile` in the `silph` subdirectory to provide a reproducible build process in an isolated environment.

```
docker build -t silph .
```

Alternatively, the build steps specified in the `Dockerfile` can be run on the host instead. Afterward, run the following commands in the `silph` subdirectory of the code repository:

```
apt-get install \
curl \
cmake \
coinor-cbc \
coinor-libcbc-dev \
cvc4 \
git \
libgmp-dev \
libboost-all-dev \
libssl-dev \
time \
zsh \
build-essential \
python3 \
python3-pip

curl --proto '=https' --tlsv1.2 \
-sSf https://sh.rustup.rs | sh -s -- -y

source "$HOME/.cargo/env"

python3 driver.py --all_features
mkdir -p scripts/aby_tests/tests
python3 driver.py -c
python3 driver.py -i
python3 driver.py -b
```

A.3.2 Basic Test

To test the basic functionality of SING, we perform share assignment on an MPC circuit and subsequently execute this circuit with ABY [4].

Performing share assignment First, we use the S^{SING} model with pretrained weights to generate a share assign-

[3] https://pytorch.org/get-started/locally/

ment for the Gauss benchmark. Run the following command in the root of the code repository:

```
python inference.py --checkpoint \
pretrained/share_assignment_supervised.pt \
--output share_assignment.txt \
example-gauss
```

The resulting share assignment computed by the model will be written to `share_assignment.txt`.

Executing a mixed-protocol MPC circuit Second, we execute the circuit with the generated share assignment in a mixed-protocol MPC setting with ABY [4]. We run both parties locally on the same host for simplicity.

Using ABY built with a Docker container (cf. section A.3.1): Run the following command in the repository root:

```
docker run --rm -it \
-v ./example-gauss:/circuit \
-v ./share_assignment.txt:/share_assignment.txt \
aby
```

Using ABY built on the host: Run the following command in the repository root:

```
BINARY=ABY-vendor/build/bin/aby_interpreter
CIRCUIT_DIR=example-gauss
SHARE_ASSIGNMENT_FILE=share_assignment.txt

# server
"$BINARY" -m mpc -f "$CIRCUIT_DIR" \
-s "$SHARE_ASSIGNMENT_FILE" -r 0 &

# client
"$BINARY" -m mpc -f "$CIRCUIT_DIR" \
-s "$SHARE_ASSIGNMENT_FILE" -r 1 &
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The share assignment runtime of SING outperforms the current state-of-the art Silph [3]. This is proven by experiment (E1) whose results are reported in Table 2 of our paper.

(C2): The MPC performance of share assignments generated by SING is similar to the current state-of-the-art Silph [3]. This is proven by experiment (E2) whose results are reported in Tables 6, 7, and 8 of our paper.

(C3): Training our models from scratch using ground-truth labels from Silph [3], computed Silph costs, or collected MPC performance measurements is efficient. This is proven by experiment (E3) whose results are reported in Table 3 of our paper.

(C4): Our models consider both the circuit structure and the operation types of individual nodes when approximating costs or assigning shares. This is proven by experiment (E4) whose results are discussed in section 8.7 of our paper.

A.4.2 Experiments

(E1): Share assignment benchmarks [5 human-minutes + 24 compute-hours + 5GB disk]: We compare the share assignment runtimes of Silph [3] and SING for the 9 benchmark circuits named in our paper.
Preparation: Compile the Silph framework without Docker and set up the Python virtual environment with all dependencies (cf. section A.3.1). Download the full processed dataset (`dataset-full.tar.gz`) from Zenodo and extract it in a suitable location. We refer to the extracted location by `$DATASET`.
Execution: Run the following command in the repository root:

```
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Silph without CSS
python benchmark_share_assignment.py \
--circuits-file paper_benchmark_c.txt \
--timeout 40000

# Silph with CSS
python benchmark_share_assignment.py \
--silph-selection-scheme css \
--circuits-file paper_benchmark_c.txt \
--timeout 40000 \
--output-csv share_assignment_css.csv
```

Results: The collected share assignment runtimes will be written to the files `share_assignment.csv` and `share_assignment_css.csv` for comparison with Silph and Silph+CSS respectively. These results can be compared with our results in the `benchmark-results/share-assignment` subdirectory. Specific result numbers depend on hardware and software configuration.

(E2): MPC benchmarks [5 human-minutes + 1.5 compute-hours + 5GB disk]: We run the 9 benchmark circuits named in our paper in a mixed-protocol MPC setting with share assignments either generated by Silph [3] or by SING and compare the results.
Preparation: Build the ABY framework without Docker and set up the Python virtual environment with all dependencies (cf. section A.3.1). Download the full processed dataset (`dataset-full.tar.gz`) from Zenodo and extract it in a suitable location. We refer to the extracted location by `$DATASET`.
Execution: Run the following commands in the repository root:

```
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

scripts/benchmark_combinations.sh
```

Results: The benchmark results will be written to different CSV files of the format `<MODEL_UUID>_<NETWORK_SETTING>.csv`. These CSV files can be compared with our results in the `benchmark-results/mpc` subdirectory. The mapping of model UUID to model variant (i.e., SING 1, 2, 3) is specified in `benchmark-results/plot.py`. Specific result numbers depend on hardware and software configuration.

We note that this experiment uses models trained with the clean training pipeline, which explicitly excludes the Silph benchmark circuits from the training dataset. We briefly describe this clean training pipeline below. The pipeline follows the same steps as E3 but uses slightly altered hyperparameters and specifically excludes the Silph benchmark circuits using the `-exclude-file` option with `exclude-silph.txt`.

```
wandb disabled
python generate_dataset_split.py \
--exclude-file exclude-silph.txt \
--alpha 0.5 \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Share assignment: SING 1
time python train.py --lr 0.01 --alpha 0.9

# Share assignment: SING 2
time python train.py --lr 0.001 --alpha 0.1 \
--predicted-cost --checkpoint \
pretrained/clean/cost_prediction_silph.pt

# Share assignment: SING 3
time python train.py --lr 0.001 --alpha 0.99 \
--predicted-cost --checkpoint \
pretrained/clean/cost_prediction_runtime.pt

mkdir dataset-cost-prediction \
|| rm -rf dataset-cost-prediction \
&& mkdir dataset-cost-prediction

ln -s $DATASET dataset-cost-prediction/raw

# Cost prediction: Train with Silph cost
time python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter no_all

# Cost prediction: Train with MPC runtime
time python train_cost_prediction.py \
--lr 0.001 --cost-name runtime-neon-wanfast \
--assignment-filter spn
```

(E3): Training our models [5 human-minutes + 1 compute-hour + 5GB disk]: We train our S^{SING} models *from scratch* in a supervised manner (SING 1) and in a semi-supervised manner (SING 2, SING 3). This training uses the pre-trained cost models C^{SING}. In addition, we train C^{SING} *from scratch* to approximate the Silph cost model and real runtime costs.

Preparation: Set up the Python virtual environment with all dependencies (cf. section A.3.1). Download the full processed dataset (`dataset-full.tar.gz`) from zenodo and extract it in a suitable location. We refer to the extracted location by `$DATASET`.

Execution: Run the following commands in the repository root:

```
wandb disabled
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Share assignment: SING 1
time python train.py --lr 0.01 --alpha 0.9

# Share assignment: SING 2
time python train.py --lr 0.001 --alpha 0.1 \
--predicted-cost --checkpoint \
pretrained/cost_prediction_silph.pt

# Share assignment: SING 3
time python train.py --lr 0.01 --alpha 0.9 \
--predicted-cost --checkpoint \
pretrained/cost_prediction_runtime.pt

mkdir dataset-cost-prediction \
|| rm -rf dataset-cost-prediction \
&& mkdir dataset-cost-prediction

ln -s $DATASET dataset-cost-prediction/raw

# Cost prediction: Train with Silph cost
time python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter silph_and_random

# Cost prediction: Train with MPC runtime
time python train_cost_prediction.py \
--lr 0.001 --cost-name runtime-neon-wanfast \
--assignment-filter silph_and_perturb
```

Results: The last five commands will print out the execution time of the respective training run. These can be compared with our results reported in Table 3 of our paper. Specific result numbers depend on hardware and software configuration.

(E4): Ablation [5 human-minutes + 1 compute-hour + 5GB disk]: We train our S^{SING} and C^{SING} models identically to (E3), with the exception that we apply two types of ablation: removing all edges and zeroing out all node features.
Preparation: Identical to preparation of (E3).
Execution: Run the following commands in the repository root:

```
wandb disabled
python generate_dataset_split.py \
--dataset-dir $DATASET

mkdir dataset || rm -rf dataset \
&& mkdir dataset

ln -s $DATASET dataset/raw

# Share assignment: normal
python train.py --lr 0.01 --alpha 0.9

python eval.py
python benchmark_mpc.py --mode benchmark \
--hashes-file ablation_hashes.txt \
--network-setting LAN \
--output-csv normal.csv \
--timeout 300

# Share assignment: remove all edges
python train.py --lr 0.01 --alpha 0.9 \
--delete-edges

python eval.py
python benchmark_mpc.py --mode benchmark \
--hashes-file ablation_hashes.txt \
--network-setting LAN \
--output-csv delete-edges.csv \
--timeout 300

# Share assignment: zero out node features
python train.py --lr 0.01 --alpha 0.9 \
--zero-features

python eval.py
python benchmark_mpc.py --mode benchmark \
--hashes-file ablation_hashes.txt \
--network-setting LAN \
--output-csv zero-features.csv \
--timeout 300

mkdir dataset-cost-prediction \
|| rm -rf dataset-cost-prediction \
&& mkdir dataset-cost-prediction

ln -s $DATASET dataset-cost-prediction/raw



# Cost prediction: remove all edges
python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter silph_and_random \
--delete-edges

python eval_cost_prediction.py

# Cost prediction: zero out node features
python train_cost_prediction.py \
--lr 0.001 --cost-name silph \
--assignment-filter silph_and_random \
--zero-features

python eval_cost_prediction.py
```

Results: The invocations of `eval.py` and `eval_cost_prediction.py` will print out accuracy (for S^{SING}) and mean-squared error metrics (for C^{SING}). The invocations of `benchmark_mpc.py` will output MPC runtimes for the ILP benchmark circuit mentioned in our paper in `normal.csv`, `delete-edges.csv`, and `zero-features.csv`. These results can be compared with the results we provide in `benchmark-results/ablation`. Specific result numbers depend on hardware and software configuration.

A.5 Notes on Reusability

Our framework can immediately be reused for fast hybrid share assignment using the provided pretrained models. Moreover, models can be retrained for new deployment settings or state-of-the-art protocol suites (e.g., [5]) by generating fresh training data from MPC runtime benchmarks and the associated optimal Silph assignments.

In the fully bootstrapped setting, SING 3 trains directly on real-world runtimes. We found this regime to be brittle: it generalized poorly, produced a negative R^2, and required substantial hyperparameter tuning and dataset refinement. Since SING 2 performed well with a simpler cost model, these results suggest that the runtime-based cost surrogate is the main bottleneck. Future work could improve this regime by constructing larger and more diverse datasets, thereby yielding a more reliable runtime surrogate. Additional directions include tighter integration with Silph [3], for example, by using S^{SING} to provide warm starts for ILP solving; evaluating stronger graph architectures, such as graph transformers [7]; or exploring alternative training paradigms, such as reinforcement learning or generative graph models [2, 6].

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/usenixsec2026/.

References

[1] Jason Ansel et al. Pytorch 2: Faster machine learning through dynamic python bytecode transformation and graph compilation. In *ASPLOS*, 2024.

[2] Aleksandar Bojchevski, Oleksandr Shchur, Daniel Zügner, and Stephan Günnemann. Netgan: Generating graphs via random walks. In *ICML*, 2018.

[3] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. Silph: A framework for scalable and accurate generation of hybrid MPC protocols. In *S&P*, 2023.

[4] Daniel Demmler, Thomas Schneider, and Michael Zohner. ABY - A framework for efficient mixed-protocol secure two-party computation. In *NDSS*, 2015.

[5] Radhika Garg, Kang Yang, Jonathan Katz, and Xiao Wang. Scalable mixed-mode MPC. In *S&P*, 2024.

[6] Ian J. Goodfellow et al. Generative adversarial nets. In *NeurIPS*, 2014.

[7] Ladislav Rampásek, Michael Galkin, Vijay Prakash Dwivedi, Anh Tuan Luu, Guy Wolf, and Dominique Beaini. Recipe for a general, powerful, scalable graph transformer. In *NeurIPS*, 2022.



USENIX Security '26 Artifact Appendix: PICS: Private Intersection over Committed (and reusable) Sets

Aarushi Goel
Rutgers University

Peihan Miao
Brown University

Phuoc Van Long Pham
Brown University

Satvinder Singh
Purdue University

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation for the protocol in the paper "PICS: Private Intersection over Committed (and reusable) Sets". We re-implement most cryptographic primitives such as OKVS, VOLE and FRI proof here, all with parallelization using the rayon package in Rust.

A.2 Description & Requirements

All tests are ran on 2 machines: one for Sender and one for Receiver. We run our experiments on 2 AWS EC2 instances, with instance type `c5a.16xlarge`. The command to run the experiments can be found on the README file.

A.2.1 Security, privacy, and ethical concerns

Running the program requires quite a lot of RAM. For machines with around 32*GB* of RAM, running the experiments for set size 2^{20} should not be an issue. If there is not enough RAM, the program will get terminated, which may crash the machine. Besides that, there is no other problem.

A.2.2 How to access

We provided a Zenodo URL for our artifact submission: https://zenodo.org/records/17958838. However, as this artifact would likely be an evolving software, we also provide the github link here, and developments should be reflected on the github repository: https://github.com/phuocchubeo123/PICS.git. The code has been tested with Rust 1.84.0.

A.2.3 Hardware dependencies

As the development process was entirely on x86 machine (AWS EC2 c5/c5a instances), it would be recommended to run this library on an x86 machine.

A.2.4 Software dependencies

The software is can be readily ran with Rust installed. However, we use some evolving libraries that might not be backward-compatible, so we provide the list of Github commits for each of these libraries in this section:

- `lambdaworks`: Version `0.14.0`.
- `rand`: Our current code works with version `0.8`. There has been a big refactor in version `0.10` that is not backward-compatible with this code.
- `rand_chacha`: Version `0.9.0`

A.2.5 Benchmarks

In the current version of the artifact, one fresh random seed is shared among two parties at the beginning of the protocol. Then two parties generate two synthetic sets with this seed. The running time of our protocol is quite deterministic given set size, with little variance.

A.3 Set-up

Throughout the development process, we follow the Rust installation guide here: https://www.digitalocean.com/community/tutorials/install-rust-on-ubuntu-linux.

A.3.1 Installation

The evaluator can simply git clone the library and run the binary file with the command in README.

A.3.2 How to run the protocol

By either having two terminals opened, or running on two different machines, the user can run the following command:

```
cargo run --release --bin oprf
(role) (address) (port)
(log_size) (num_threads) (LPN_param)
(sender_commit) (receiver_commit)
```

Each parameter has the following meaning:

- `role`: The role of this machine, either `sender` or `receiver`.

- `address`: For both machines, set this parameter as the IP address of the *receiver*, as we designate the receiver as the TcpListener.
- `port`: The port that is used for communication.
- `log-size`: The logarithm base 2 of two sets' size. For example, `log-size` = 4 means the set size is 16. We recommend setting the set sizes of two parties to be equal.
- `threads`: Number of threads for parallelization.
- `params`: Choosing the LPN parameter for running VOLE. We currently support three sets of LPN parameters: LPN17, LPN21, and LPN25. A parameter set of *LPNXX* means that one can use 2^{XX} VOLE instances, before needing to run `extend` again for another set of fresh 2^{XX} VOLE instances.
- `sender-commit` / `receiver-commit`: Either 0 or 1, to set whether a party runs the committing phase or not.

A.3.3 Basic Test

A run for sets of size 2^{16} would take less than 5 seconds, and should confirm the functionality of the artifact.

A.4 Evaluation workflow

A.4.1 Major Claims

The main major claim of this paper is small commitment size and running time almost linearly dependent on the two sets' size. We tracked the communication cost and print it out when running the binary to support the first claim. For the second claim, we refer to the table in the submitted paper.

A.4.2 Experiments

The following two commands are examples to run the protocol for set sizes 2^{20}, where both sender and receiver commit to their sets. This example experiment runs in the LAN setting, where both parties talk to each other through localhost. Each party parallelizes the computation through 4 cores. First, run on the receiver machine:

```
cargo run --release --bin oprf
receiver 127.0.0.1 8080 20 4 1 1 1
```

Secondly, run on the sender machine:

```
cargo run --release --bin oprf
sender 127.0.0.1 8080 20 4 1 1 1
```

The user should run the receiver's machine first, so that the receiver could set up a Tcp connection, before the sender could connect to it.

Currently, we have computed three sets of LPN parameters, that would work best with the corresponding set sizes provided in Table 1.

Set size	LPN Parameters	Option (`LPN_param`)
2^{16}	`LPN17`	0
2^{20}	`LPN21`	1
2^{21}	`LPN25`	2

Table 1: Suitable LPN parameters for experiments

A.4.3 Setting up the WAN experiment

We use the following script: https://github.com/phuocchubeo123/setup-wan-network.git, which uses the Linux `tc` command to set up the bandwidth (Megabits) and latency (milliseconds) for the experiment.

A.4.4 On choosing LPN parameters

We run a grid search to choose LPN parameters, by running the following repo: https://github.com/phuocchubeo123/LPN-Estimator-Multiprocess.git. The python code that calculates the security level of a set of LPN parameters can be run with the following example command:

```
python home/estimator.py
N=1024 k=652 t=57 q=12 rgular
```

Here, `N`, `k` and `t` are LPN parameters. The field bit size is `q`, for example if `q` $= 12$ then the field is $\mathbb{F}_p$ where $\log p \geq 12$.

The types of noise for LPN that one can choose are `exact` and `rgular`. Here, due to a regex parsing error, one need to use `rgular` instead of `regular` for regular LPN noise. This parsing error appears from the original repo: https://github.com/RabbitCabbage/LPN-Estimator.git. The authors modify this repo to run the security parameter calculation with multiple processes, to speed up the search.

A.5 Notes on Reusability

In the beginning, the authors hope that this artifact would be the first implementation of many cryptographic primitives in Rust. However, we are made aware later that `swanky`[1] has implemented a lot of the cryptographies in Rust. The authors are contemplating on whether to rewrite many portions of this artifact to use swanky directly, and integrate into swanky themselves, to keep this library longer for other researchers in the PSI community to use. If revamping happens, most likely the running time would be faster than the current version of the artifact.

[1] https://github.com/GaloisInc/swanky.git

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: FLOSS: Fast Linear Online Secret-Shared Shuffling

Ian Chang
University of Washington

Sela Navot
University of Washington

Alex Ozdemir
Max Planck Institute for Security and Privacy, Georgia Tech

Nirvan Tyagi
University of Washington

A Artifact Appendix

A.1 Abstract

We present FLOSS, a protocol for fast online linear secret-shared shuffling and a framework to build arithmetic permutation circuits. Our artifact includes:

1. the computational and communication efficiency comparison of FLOSS against various shuffling baselines;
2. the computational and communication efficiency evaluation of FLOSS used in radix sort (Radix[FLOSS]) compared with other state-of-the-art sorting alternatives;

This artifact reproduces the tables and figures in our paper.

A.2 Description & Requirements

FLOSS enables efficient online shuffling in the two-party computation (2PC setting). We provide shuffling benchmarks for FLOSS, the constant round malicious secure permutation network by Mohassel et al. (PermNet), Oblivious Punctured Matrix from Song et al. (OPM), and the canonical $\log(n)$-round malicious-secure permutation network circuit from Waksman (SimplePermNet). We also provide a case-study applying our shuffling protocol to several sorting benchmarks. These sorting benchmarks include MP-SPDZ's sorting network (SortNet), MP-SPDZ quicksort combined with shuffling protocols (StS[FLOSS], StS[OPM], StS[PermNet], StS[SimplePermNet]), and radix-sort combined with shuffling protocols (Radix[FLOSS], Radix[PermNet], Radix[SimplePermNet]). We evaluate both the preprocessing time (MP-SPDZ preprocessing generation, any input-independent circuit computation) and the online time (any input-dependent circuit computation).

We highly recommend using AWS EC2 instances for the artifact evaluation, which requires approximately $150 cost in AWS credits. The minimal hardware requirements to run our full set of our benchmarks is two AWS c5.18xlarge instances (72 vCPUs and 144GB memory each), each running Ubuntu 22.04 LTS. We provide one-run scripts that automatically instantiates the AWS instances with required dependencies and benchmarks.

For reviewers without AWS access, we also support running the *full* benchmark suite, including the pareto plots, on local hardware comparable to the AWS setup: either two local Ubuntu servers (each with ≈72 hardware threads and 144GB memory, connected over a high-bandwidth network), or a single large local Ubuntu server (with ≈144 hardware threads and 288GB memory) partitioned into two Docker containers that emulate the two parties. We provide a Dockerfile and container scripts that automate installation and benchmarking for this setup.

Ubuntu 22.04 LTS (recommended) or Mac OS X Sequoia is required in order to run benchmarks on a single smaller local computer, which only runs a *subsample* of our benchmarks due to decreased compute power. We require that the user has at least the compute power of at least 16 GB of memory and an Apple M4 chip for CPU (or equivalent).

A.2.1 Security, privacy, and ethical concerns

FLOSS is a protocol for privacy-preserving computation. It can be used to compute any function over user data in the malicious-secure two-party setting.

Note that there are always challenges when migrating from an existing system to a privacy-compliant one. MPC-based systems are still very new, and they are not easy to implement correctly. As a result, companies and organizations should think carefully about the privacy implications of the statistics they are computing over user data and be transparent about what aggregated data is being computed.

A.2.2 How to access

FLOSS is available here. Reviewers should download and extract the Zenodo archive, and then run all commands be-

low unless otherwise stated. The full artifact documentation, including `ARTIFACTS.md`, is included in the Zenodo archive.

We have also prepared scripts to automatically launch clusters and run the experiments, given that the reviewer has access to AWS EC2 instances, as well as Docker-based scripts for reviewers who instead have access to comparable local server hardware. The essential setup and reproduction steps for all options are included directly below.

A.2.3 Hardware dependencies

FLOSS does not require any specialized hardware. Our experiments were performed on AWS c5.18xlarge instances, each with 72 vCPUs and 144 GB memory. Reviewers without AWS access can reproduce the full benchmark suite on comparable local hardware instead: either two local Ubuntu servers, each with approximately 72 vCPUs and 144GB memory, or a single large local Ubuntu server with at least 144 vCPUs and 288GB memory, which we partition into two Docker containers of 72 vCPUs and 144GB memory each to emulate the two parties. Different hardware configurations will affect the concrete performance of FLOSS, but will result in a similar performance gain over the baselines.

A.2.4 Software dependencies

Option 1 (AWS EC2, recommended, full results). The one-run scripts in `artifacts/` automatically set up the AMI cluster, install prerequisites on the remote machines, execute the full benchmark suite, and collect the results. The local machine used to control the AWS run needs Python 3, the Python plotting libraries, TeX Live/MacTeX with `pgf` and `xcolor`, AWS CLI version 2, and AWS credentials configured for region `us-west-2` with JSON output.

Option 2 (large local server, no AWS, full results). This option reproduces the full benchmark suite, including the pareto plots, without AWS access, using two local Ubuntu servers or a single large local Ubuntu server running Docker. We provide `artifacts/local-large.Dockerfile`, which clones FLOSS, installs the OS, Python, Rust, and MP-SPDZ dependencies (including the `-DINSECURE` MP-SPDZ build described below), installs OPM's dependencies, and builds FLOSS inside the image. The local server needs Docker installed with enough free vCPUs and memory to allocate 72 vCPUS and 144GB, respectively, to each of two containers (288GB total) if using the single-server Docker setup; the two-server setup instead needs FLOSS and its dependencies installed natively on each server, as in Option 3 below.

Option 3 (single local computer, subsampled results). To install FLOSS only on a smaller local machine, the OS required is either Ubuntu 22.04 LTS or Mac OS X (Sequoia 15.7.4 works to our knowledge). This workflow only runs a subsample of the experiment plots, so we require that the user has at least 16GB RAM and an Apple M4 chip or equivalent. The user must set up the following dependencies:

1. Install necessary libraries (Ubuntu): `sudo apt-get install automake build-essential clang cmake git libboost-dev libboost-filesystem-dev libboost-iostreams-dev libboost-thread-dev libgmp-dev libntl-dev libsodium-dev libssl-dev libtool python3 unzip`
2. Install necessary libraries (Mac OS X): `brew install automake cmake boost gmp ntl libsodium openssl libtool python unzip`
3. Install Python libraries: `python3 -m pip install numpy matplotlib pandas scipy latex`
4. Install TexLive/MacTeX: https://tug.org/texlive/
5. Install TexLive/MacTeX libraries: `sudo tlmgr update -self; sudo tlmgr install pgf; sudo tlmgr install xcolor`
6. Install Rust: https://rust-lang.org/
7. Download and extract the Zenodo archive, then enter the extracted project root.
8. Compile MP-SPDZ and the 2PC malicious-secure protocols: `cd mp-spdz-0.4.2 && echo "MY_CFLAGS += -DINSECURE" » CONFIG.mine && make clean && make setup && make -j8 pairwise-offline.x mascot-offline.x lowgear-party.x mascot-party.x`. The `-DINSECURE` flag enables MP-SPDZ's insecure benchmarking functionality for local key generation; it is required for the artifact benchmarks (MP-SPDZ will otherwise report `You are trying to use insecure benchmarking functionality for local key generation`) and should not be used for production deployments. If you add the flag after a previous MP-SPDZ build, run `make clean` again before recompiling.

A.2.5 Benchmarks

We provide benchmarks for the following tasks.

1. Computation time and communication costs comparison for shuffling input sizes $2^{10}, 2^{12}, 2^{14}, 2^{16}$ on a single smaller local computer (Option 3) and input sizes $2^{10}, 2^{12}, 2^{14}, 2^{16}, 2^{18}, 2^{20}$ on two AWS machines (Option 1) or two comparable large local machines/containers (Option 2). The supported shuffling benchmarks are FLOSS, PermNet, SimpleNet, and OPM.

2. Computation time and communication costs comparison for sorting 32-bit inputs with sizes $2^9, 2^{10}$ on a single smaller local computer (Option 3) and input sizes $2^9, 2^{10}, 2^{11}, 2^{12}, 2^{13}$ on two AWS machines (Option 1) or two comparable large local machines/containers (Option 2). The supported sorting benchmarks are Radix[FLOSS], Radix[PermNet], Radix[SimpleNet], StS[FLOSS], StS[PermNet], StS[SimpleNet], StS[OPM], and SortNet.

3. Offline monetary cost vs. online time single-point plot summary for shuffling 2^{20} inputs and sorting 2^{13} inputs, which requires the full-size data collected on two AWS machines (Option 1) or two comparable large local machines/containers (Option 2).

A.2.6 Limitations

The full paper plots require either two AWS `c5.18xlarge` instances (Option 1) or comparable local hardware (Option 2, two 72-vCPU/144GB servers or a single 144-vCPU/288GB server split into two Docker containers), and approximately 24 hours of compute time. The Option 3 single-computer workflow is intended as a functional and partial-results check: it omits the largest shuffle size 2^{20} and sorting size 2^{13} to avoid exceeding memory on a 16GB machine. Consequently, Option 3 cannot reproduce the offline monetary cost vs. online time Pareto plots; reviewers who wish to reproduce those plots should use the AWS workflow (Option 1) or the large local server workflow (Option 2). Reviewers using the single-server Docker variant of Option 2 should also be aware that both simulated parties share the same physical CPU, memory, and disk, so results may differ somewhat from a genuine two-machine deployment (Option 1 or the two-server variant of Option 2); we mitigate this by pinning each container to a disjoint set of CPU cores and memory and by rate-limiting the virtual network to 25 Gbps to match the AWS network bandwidth.

A.3 Set-up

In this section, we provide information about setting up and running the artifacts.

A.3.1 Installation

All setup starts by downloading and extracting the Zenodo archive.

Option 1 (AWS).

1. Install TeX Live/MacTeX, Python 3, the Python packages `numpy`, `matplotlib`, `pandas`, `scipy`, and `latex`, and AWS CLI version 2 on the personal host machine.

2. Configure AWS with `aws configure`, using `us-west-2` as the default region and `json` as the default output format.

3. Start the two-machine cluster from the project root with `python3 artifacts/start_cluster.py`.

Option 2 (large local server).

1. From the project root, build the provided Docker image, which installs all dependencies and builds FLOSS inside the container: `docker build -memory=8g -memory-swap=10g -t floss-large-local -f artifacts/local-large.Dockerfile .`

2. Create a Docker bridge network for the two simulated parties: `docker network create -driver bridge -subnet 172.28.0.0/16 floss-bench-net`.

3. Start one container per party: `docker run -dit -name floss-party0 -cpuset-cpus="0-71" -memory="144g" -shm-size="16g" -cap-add=NET_ADMIN -privileged=true -network floss-bench-net -ip 172.28.0.2 floss-large-local bash`, and similarly `docker run -dit -name floss-party1 -cpuset-cpus="72-143" -memory="144g" ... -ip 172.28.0.3 floss-large-local bash`.

4. Restrict each container's network to 25 Gbps to match the AWS setting: `docker exec floss-party0 bash -lc "tc qdisc replace dev eth0 root tbf rate 25gbit burst 32mb latency 50ms"` (and likewise for `floss-party1`).

5. Create the `parties.txt` config file inside both containers for the Rust benchmarks, e.g. `docker exec floss-party0 bash -lc "cd /root/repo && printf '172.28.0.2:8644\n172.28.0.3:8645\n' > parties.txt"` (repeated for `floss-party1`).

6. (Alternative) If instead using two separate physical local servers, install FLOSS and its dependencies natively on each server as in Option 3, and create the same `parties.txt` on both using an existing local network bridge between the two servers.

Option 3 (single local computer).

1. Install the OS libraries, Python, TeX, and Rust dependencies listed in the Software Dependencies section.

2. Compile MP-SPDZ from the project root with `cd mp-spdz-0.4.2 && echo "MY_CFLAGS += -DINSECURE" » CONFIG.mine && make clean && make setup && make -j8 pairwise-offline.x mascot-offline.x lowgear-party.x`

`mascot-party.x` (the `-DINSECURE` flag enables the insecure local key generation needed by the benchmarks; see the Software Dependencies section for details).

A.3.2 Basic Test

After setup, one can quickly check the artifact's functionality by running one small FLOSS benchmark test.

Option 1 (AWS).

1. Run `python3 artifacts/start_cluster.py`; this creates the two EC2 instances and copies the required configuration files, including `parties.txt`, to both machines.
2. Run `python3 artifacts/run_floss_bench.py`; a few synchronous tests will complete in roughly 5–10 minutes before the long-running benchmarks are launched in the background.

Option 2 (large local server).

1. After the two containers are running, open two terminal tabs, one for each party.
2. On party 0, run `docker exec -it floss-party0 bash -c "cd /root/repo && env ALONE=false RANK=0 IP_FILE=parties.txt cargo bench -bench floss_shuffle"`.
3. At approximately the same time, on party 1, run the same command with `RANK=1` on `floss-party1`.

Option 3 (single local computer).

1. Run `cargo bench -bench floss_shuffle`; completion produces FLOSS shuffle times and confirms that the Rust and MP-SPDZ components are usable.

For all options, verify beforehand that `parties.txt` has been created correctly in the project root on *both* parties (each entry gives that party's network address and port)–an incorrect or missing `parties.txt` leads to connection failures between the two parties.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): FLOSS achieves an online computation time speedup of 800× and communication cost speedup of 6× over state-of-the-art alternatives for 2PC shuffling over state-of-the-art alternatives. This is proven by experiment (E1). The result is described in Section 7.2 and is illustrated in Table 1, Figure 7, and Figure 8.

(C2): FLOSS achieves an offline monetary cost reduction of 2× for 2PC shuffling over state-of-the-art alternatives. This is proven by experiment (E1). The result is described in Section 7.3 and is illustrated in Figure 7.

(C3): FLOSS requires ≈ 2× less SPDZ preprocessing than PermNet for 2PC shuffling. This is proven by experiment (E1). The result is described in Section 7.3 and is illustrated in Table 3.

(C4): Radix[FLOSS] achieves an online computation time speedup of 5× and communication cost speedup of 3× for sorting over state-of-the-art alternatives. This is proven by experiment (E2). The result is described in Section 7.4 and is illustrated in Figure 9 and 10.

(C5): Radix[FLOSS] achieves an offline monetary cost reduction of 3× for sorting over state-of-the-art alternatives. This is proven by experiment (E2). The result is described in Section 7.4 and is illustrated in Figure 9.

A.4.2 Experiments

(E1): [Shuffling baseline comparisons] [1 human-hour + 24 compute-hours (running full tests on AWS) or 3 compute-hours (running subsampled tests locally) + 256GB disk]: This experiment will describe the offline and online costs for the shuffling baselines. It will automatically run the shuffling (and sorting) benchmarks, and they will automatically collect both the offline/online times and communication cost into individual CSV files. This experiment will specify the plotting scripts and output files needed to validate the main *shuffling* claims from the paper.

Preparation: Complete the AWS (Option 1), large local server (Option 2), or single-computer (Option 3) setup from Section A.3. For the full AWS run, configure AWS credentials on the host machine and run `python3 artifacts/start_cluster.py` from the project root. For the large local server run, build `artifacts/local-large.Dockerfile`, start the two party containers, and create `parties.txt` on both, as described in Section A.3. For the local subsampled run, compile MP-SPDZ and ensure Docker is available for the OPM baseline.

Execution: For the full AWS run, execute the benchmark suite with `python3 artifacts/run_floss_bench.py`. The initial setup takes approximately 5–10 minutes and may require accepting prompts with `Y`; after that, the benchmarks run asynchronously under `nohup`. Wait approximately 24 hours, then collect results with `python3 artifacts/collect_floss_bench_results.py`. After collecting results, tear down the cluster with `python3 artifacts/teardown_cluster.py`.

For the large local server run, open one terminal per party and, at approximately the same time, run on

party 0 (e.g. `docker exec -it floss-party0 bash`, then `cd /root/repo`): `env ALONE=false RANK=0 IP_FILE=parties.txt cargo bench -bench <name>` for each of `simple_perm_network_shuffle`, `perm_network_shuffle`, `floss_shuffle`, `sort_with_simple_perm_network`, `sort_with_floss`, `sort_with_perm_network`, `sort_with_quicksort`, and `sort_with_sorting_network`, followed by `./scripts/bench_opmcc.sh 0 0 172.28.0.2:39530,172.28.0.3:39531`; run the identical commands with `RANK=1` on `floss-party1`, with the OPM script argument `./scripts/bench_opmcc.sh 0 1 172.28.0.2:39530,172.28.0.3:39531` (substitute your containers'/servers' actual IPs if different). If any individual benchmark fails partway, it can be re-run in isolation. This takes approximately 24 compute-hours, comparable to Option 1. Afterwards, collect the output CSVs from party 0 onto the host: `docker exec floss-party0 bash -lc "mkdir -p /root/results && cp /root/repo/*.csv /root/results/"` then `docker cp floss-party0:/root/results/. .` from the project root.

For the local subsampled run, build and run the OPM baseline in Docker with `cd OPM/`, `docker build -t mosac:latest .`, `docker run -it -name mosac-dev -cap-add=NET_ADMIN -privileged=true mosac:latest bash`, and then `./scripts/bench_opmcc.sh 1` inside the container. Exit Docker and copy `shuffle_opmcc_offline.csv` and `shuffle_opmcc_online.csv` from the container path `mosac-dev:/opm/` to the project root using `docker cp`. Then run the remaining local benchmarks: `cargo bench -bench simple_perm_network_shuffle`, `cargo bench -bench perm_network_shuffle`, `cargo bench -bench floss_shuffle`, `cargo bench -bench sort_with_simple_perm_network`, `cargo bench -bench sort_with_floss`, `cargo bench -bench sort_with_perm_network`, `cargo bench -bench sort_with_quicksort`, and `cargo bench -bench sort_with_sorting_network`.

Results: Generate plot data from the project root by running `cd plots/`, `python3 gen_plot_data.py`, `pdflatex main.tex`, and `cd ..` for the AWS (Option 1) or large local server (Option 2) workflow; this can be run on the host after installing the plotting dependencies, or inside the party-0 container/server. For the local subsampled workflow (Option 3), use `python3 gen_plot_data.py -alone 1` instead. The expected shuffling outputs are `plots/plot_data/shuffle_offline_time.csv`, `plots/plot_data/shuffle_online_time.csv`, `plots/main.pdf`, `shuffle_floss_offline.csv`, and `shuffle_perm_network_offline.csv`. To interpret the results with the main claims:

- `(Experiment E1) Table 1 + Individual shuffling data` provides the online and offline computation and communication detailed costs for all the shuffling baselines for Claim C1. FLOSS is expected to have the fastest online computation time and lowest online communication cost over all shuffling baselines. There is an approximate online time speedup of 800× and communication cost speedup of 6×.
- `(Experiment E1 + E2) Figures 7-10` provides the offline/online cost shuffling trend plots for Claim C1 and the offline monetary cost vs. online time shuffling plot for Claim C2. FLOSS has 2× lower offline monetary cost than all other baselines.
- `(Experiment E1) Table 3 (in Project Root directory)` provides the offline preprocessing breakdown costs for Claim C3. FLOSS has ≈ 2× less SPDZ preprocessing than PermNet for 2PC shuffling.

(E2): [Sorting baseline comparisons] [1 human-hour + 24 compute-hours (running full tests on AWS) or 3 compute-hours (running subsampled tests locally) + 256GB disk]: This experiment will describe the offline and online costs for the sorting baselines. It will automatically run the sorting (and shuffling) benchmarks, and they will automatically collect both the offline/online times and communication cost into individual CSV files. This experiment will specify the plotting scripts and output files needed to validate the main *sorting* claims from the paper. **Note that if you already ran the "Preparation" and "Execution" stages in Experiment E1, do not re-run either of these stages for Experiment E2. Only run the "Results" stage since all results are collected already.**

Preparation: Use the same preparation as in Experiment E1. If E1 has already been prepared and executed, do not repeat preparation or execution–the E1 workflow collects the sorting data needed for E2 as well.

Execution: Use the same execution as in Experiment E1 **if not performed Experiment E1 execution yet.** For the full AWS run, execute `python3 artifacts/run_floss_bench.py`, wait approximately 24 hours, and collect results with `python3 artifacts/collect_floss_bench_results.py`; tear down the cluster afterwards with `python3 artifacts/teardown_cluster.py`. For the large local server run, run the same per-party `cargo bench` and `bench_opmcc.sh` commands listed under E1 on both parties simultaneously, then collect the CSVs from party 0 as in E1. For the local subsampled run, run the Docker-based OPM commands and the local

`cargo bench` commands listed under E1. The sorting benchmarks are `sort_with_simple_perm_network`, `sort_with_floss`, `sort_with_perm_network`, `sort_with_quicksort`, and `sort_with_sorting_network`.

Results: Generate plot data as in E1: `cd plots/`, then `python3 gen_plot_data.py` for AWS or large local server results, or `python3 gen_plot_data.py -alone 1` for local subsampled results, followed by `pdflatex main.tex`. The expected sorting outputs are `plots/plot_data/sort_offline_time.csv`, `plots/plot_data/sort_online_time.csv`, and `plots/main.pdf`. To interpret the results with the main claims:

- `(Experiment E2) Individual sorting data` provides the online and offline computation and communication detailed costs for all the sorting baselines for Claim C4. Radix[FLOSS] is expected to have the fastest online computation time and lowest online communication cost over all sorting baselines. There is an approximate online time speedup of 5× and communication cost speedup of 3×.
- `(Experiment E1 + E2) Figures 7-10` provides the offline/online cost sorting trend plots for Claim C4 and offline monetary cost vs. online time sorting plot for Claim C5. Radix[FLOSS] has 3× lower offline monetary cost than all other baselines.

A.5 Notes on Reusability

FLOSS is well suited for performing privacy-preserving computations in the malicious-secure 2PC setting. Past malicious-secure shuffling protocols have been well studied in the malicious-secure 3PC honest-majority setting; however, this requires setting up an additional honest third party beyond the two-party case.

FLOSS is flexible, such that any future developer who wishes to add their own privacy-preserving protocol can do so easily within the framework. The implementation is extensible, allowing developers to define additional higher-level custom gates on top of the gates we provide, making it easy to reuse and construct more complex interactive arithmetic permutation circuits. For instance, the sorting case study demonstrates the power of using FLOSS's arithmetic permutation circuit framework.

An exciting future direction is to implement more complicated privacy-preserving analytics protocols through FLOSS, such as database joins and heavy hitters.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: LPG: Raise Your Location Privacy Game in Direct-to-Cell LEO Satellite Networks

Quan Shi[1], Liying Wang[2]

A Artifact Appendix

A.1 Abstract

Multi-tenant direct-to-cell (D2C) Low Earth Orbit (LEO) satellite networks pose significant risks to users' location privacy by linking Mobile Network Operator (MNO)- managed identities with Satellite Network Operator (SNO)- visible locations. We present LPG, the first protocol-layer framework that decouples identity from location while maintaining essential functions like billing and policy in resource-constrained orbital environments.

This artifact provides the implementation of the LPG protocol, including the source code for the User Equipment (UE), Satellite Network Operator (SNO), and Mobile Network Operator (MNO) components. The paper's evaluation involves a real in-orbit LEO satellite. Since this hardware is physically inaccessible for public or reviewers, this artifact focuses on the functional correctness of the protocol logic. We also provide raw telemetry logs extracted from the satellite flight to substantiate the paper's specific hardware claims.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

We make our research artifacts accessible in `https://zenodo.org/records/17906146` (Public Version) and `https://zenodo.org/records/18428501` (Restricted Version).

A.2.3 Hardware dependencies

The LPG protocol can be evaluated on standard commodity hardware.

- Processor: The paper's evaluation results (latency, power, temperature) were collected on the BUPT-3 satellite payload (8-core ARM processor, 16 GB RAM). An ARM64 Linux environment (e.g., Raspberry Pi 4/5 or an ARM-based VM) is recommended.

A.2.4 Software dependencies

The artifact is tested on Ubuntu 20.04 LTS and Ubuntu 22.04 LTS. It requires a hybrid build environment for C++, Rust, and Go.

- OS: Linux (Ubuntu 20.04/22.04).
- Compilers & Build Tools:
  - **C++** Compiler supporting C++17 (e.g., `g++` or `clang++`).
  - **Rust** and `cargo` (required for the SCAC module integration).
  - **Go** (version 1.18+, required for ZKLP bridge compilation).
  - `CMake` ($\geq$ 3.10) and `Make`.
  - `git`.
- System Libraries:
  - OpenSSL development libraries (`libssl-dev`).
  - GMP development libraries (`libgmp-dev`).
  - `pkg-config`.
- Python: Python 3 and Jupyter Notebook are required to regenerate the plots using `scripts/plot.ipynb`.

A.2.5 Benchmarks

The artifact includes both benchmarks for protocol verification and real-world telemetry datasets collected from the BUPT-3 LEO satellite in-orbit deployment.

- Protocol Benchmarks: The `lpg_main` (from `LPG-code`) generates synthetic user identities and tokens to execute the full LPG protocol lifecycle, including UE and SNO unlinkable token generation (DualToken), Offline Authentication and Key Agreement (OAKA), Zero-Knowledge Location Proof (ZKLP) generation that enables session management, and Scalable Collaborative Aggregate Claiming (SCAC).

- Satellite Telemetry Datasets: The Results directory contains raw execution logs and hardware telemetry (power consumption, temperature) collected from the BUPT-3 LEO satellite (COSPAR ID 2025-103F). These datasets correspond to the evaluation results reported in Section 5 of the paper.

A.3 Set-up

A.3.1 Installation

The artifact is designed for a standard Linux environment (Ubuntu 20.04+ recommended). The build process requires a hybrid environment for C++, Rust, and Go.

1. Install System Dependencies

```
sudo apt update
sudo apt install -y libssl-dev pkg-config \
    libgmp-dev
```

2. Update and Integrate Submodules

```
git submodule update --init --recursive
sh LPG-code/appy-integration.sh
```

3. Build and Run

```
cd LPG-code
mkdir build && cd build
cmake ..
make
./lpg_main
```

Upon success, the `lpg_main` binary is generated in the `build` directory.

A.3.2 Basic Test (Liying)

`lpg_main` integrates simple tests for basic functionalities. The expected successful output is shown in Figure 1.

A.4 Evaluation workflow

A.4.1 Major Claims

The major claims of our paper are that LPG provides a functional, privacy-preserving protocol framework for Direct-to-cell D2C LEO networks, operating within the constraints of satellite hardware. Since the computational tasks running on SNO are conducted in a real space environment, the results you obtain when running the code on the server may differ from those presented in the paper.

(C1): Efficient Token Registration. LPG's DualToken mechanism achieves practical generation times ($\sim$ 33ms for $\mathsf{Tok}_{\mathsf{UE}}$) and storage overhead comparable to commercial SIM cards. This is validated by experiment (E1) and discussed in §5.3.1 (Figures 12 (b)(c)).

```
(base) ➜  LPG-code ./build/lpg_main
===== LPG System Integration Test =====

------ Testing Dual Token Mechanism ------
Dual Token Mechanism Test: PASSED

------ Testing OAKA Protocol ------
OAKA Protocol Test: PASSED

------ Testing Session Management Table (SMT) ------
SMT Test: Session Creation PASSED
SMT Test: Session Update PASSED
SMT Test: Session Termination PASSED

------ Testing ZKLOC ------
[Go] Compiling Circuit (this happens only once)...
17:01:47 INF compiling circuit
17:01:47 INF parsed circuit inputs nbPublic=4 nbSecret=2
17:01:47 INF building constraint builder nbConstraints=25486
[Go] Running Setup (generating keys)...
ZKLOC Test: Initialization PASSED
Input Lat: 0.691891, Lng: 0.709619, Resolution: 0, I: 1, J: 0, K: 1
17:01:50 DBG constraint system solver done nbConstraints=25486 took=17.903625
17:01:50 DBG prover done acceleration=none backend=groth16 curve=bn254 nbConstraints=25486 took=102.4445
ZKLOC Test: Proof Generation PASSED
17:01:50 DBG verifier done backend=groth16 curve=bn254 took=1.987291
ZKLOC Test: Proof Verification PASSED

------ Testing SCAC (Groth16 ZK-SNARK) ------
Start:   Groth16::Generator
··Start:   Constraint synthesis
··End:     Constraint synthesis .....................................................885.834µs
··Start:   Inlining LCs
··End:     Inlining LCs .............................................................2.815ms
··Start:   Constructing evaluation domain
··End:     Constructing evaluation domain ..........................................23.459µs
··Start:   R1CS to QAP Instance Map with Evaluation
····Start:   Evaluate Lagrange coefficients
····End:     Evaluate Lagrange coefficients .......................................605.791µs
··End:     R1CS to QAP Instance Map with Evaluation .................................1.468ms
··Start:   Compute G2 table
··Start:   Compute G2 table
··End:     Compute G2 table ........................................................1.592ms
··Start:   Calculate B G2
··End:     Calculate B G2 ..........................................................2.220ms
··Start:   Compute G1 window table
··End:     Compute G1 window table .................................................1.280ms
··Start:   Generate the R1CS proving key
····Start:   Calculate A
····End:     Calculate A ...........................................................903.791µs
····Start:   Calculate B G1
····End:     Calculate B G1 ........................................................617.375µs
····Start:   Calculate H
····End:     Calculate H ...........................................................1.662ms
····Start:   Calculate L
····End:     Calculate L ...........................................................1.018ms
··End:     Generate the R1CS proving key ...........................................5.118ms
··Start:   Generate the R1CS verification key
··End:     Generate the R1CS verification key ......................................362.750µs
··Start:   Convert proving key elements to affine
··End:     Convert proving key elements to affine ..................................780.125µs
End:     Groth16::Generator .....................................................17.725ms
Start:   Groth16::Prover
··Start:   Constraint synthesis
··End:     Constraint synthesis .....................................................704.625µs
··Start:   Inlining LCs
··End:     Inlining LCs .............................................................2.547ms
··Start:   R1CS to QAP witness map
··End:     R1CS to QAP witness map .................................................4.290ms
··Start:   Compute C
··End:     Compute C ...............................................................5.026ms
··Start:   Compute A
··End:     Compute A ...............................................................1.782ms
··Start:   Compute B in G1
··End:     Compute B in G1 .........................................................1.350ms
··Start:   Compute B in G2
··End:     Compute B in G2 .........................................................3.330ms
··Start:   Finish C
··End:     Finish C ................................................................1.500µs
End:     Groth16::Prover .....................................................19.333ms
SCAC Test: ZK-SNARK Proof Verification PASSED
===== LPG System Integration Test Finished ====
```

Figure 1: Expected bash output when running `./lpg_main`

(C2): Lightweight Attachment (OAKA). The OAKA protocol successfully performs anonymous authentication with acceptable latency ($\sim$ 140ms extra latency) and low communication overhead. This is validated by experiment (E1) and discussed in §5.3.2 (Figures 14 (a)).

(C3): User-Configurable Location Disclosure (ZKLP). The ZKLP mechanism supports variable location resolutions ($\mathsf{res} \in \{0..15\}$) with consistent proof sizes ($\sim$ 430 bytes) and verification times. This is validated by experiment (E1) and discussed in §5.3.3 (Figure 13).

(C4): Scalable Settlement (SCAC). The SCAC protocol efficiently aggregates claims, with server overhead and constraint counts scaling linearly with the number of sessions. This is validated by experiment (E2) and discussed in §5.3.4 (Figures 15 (a)(b)).

(C5): Satellite Viability. The on-board components of LPG operate within the strict power (< 40W) and temperature ($< 62^{\circ}$C) budgets of LEO satellites. This is supported by

```cpp
// In LPG-code/main.cpp
// N=8
const size_t num_sessions = 8;
SCAC_SNO sno(num_sessions);
SCAC_MNO mno(sno.GetZKKeys(), sno.GetPoseidonParams());

// length of the vectors: K=4
std::vector<uint64_t> individual_data_usages = {100, 150, 50, 200};
std::vector<uint64_t> individual_session_ids = {1001, 1002, 1003, 1004};
```

Figure 2: Adjusting parameters $\mathcal{N}$ and $\mathcal{K}$

the in-orbit telemetry logs analyzed in experiment (E3) and discussed in §5.2 and §5.3.4 (Figures 15 (d)(e)(f)). This involves measurements of the satellite hardware and is not included in the logs output by our `lpg_main` code.

A.4.2 Experiments

We describe the experiments to validate the functional correctness of the LPG protocols (E1, E2) and to audit the real-world satellite execution traces (E3).

(E1): Protocol Functional Verification: Validates the cryptographic correctness and logic flow of Registration, OAKA, and ZKLP (Claims C1, C2, C3).
Preparation: Ensure all dependencies (C++, Rust, Go) are installed and the project is built successfully using `cmake` as described in the Installation section.
Execution: Run the main test.
Results: The program will sequentially execute Token Generation, Offline Authentication and Key Agreement (OAKA), and ZKLP Proof Generation.

- **Expected Outcome:** The console output should display "PASSED" for each module.
- **Note:** Execution times reported on the evaluator's local machine will differ from the ARM-based satellite COTS device results reported in the paper, but the logical correctness and computational complexity of ZKLP will be maintained.

(E2): Settlement Scalability Test: Validates the scalability of the SCAC aggregation protocol (Claim C4).
Preparation: Requires the same build environment as E1.
Execution: Run the main test.
Results: The script simulates the aggregation of multiple sessions.

- **Expected Outcome:** The program completes aggregation without error. Evaluators should observe that the processing time and constraint generation follow the linear trend described in §5.3.4, validating the scalability claim.

(E3): Satellite Telemetry Audit: Validates the real-world deployment and environmental impact claims (Claim C5) by inspecting artifacts generated in orbit.
Preparation: Locate the `Results/Satellite/` directory in the artifact package.
Execution: This is a data inspection task. Reviewers should examine the provided log files:

- Execution Logs: e.g., `at/arm-at-*.log` (OAKA execution).
- Telemetry Logs: e.g., `...temp-power.log` (Hardware sensors).

Results: Compare the timestamps in the execution logs with the telemetry logs.

- **Expected Outcome:** The logs confirm that the experiments were conducted on the BUPT-3 satellite platform. The power and temperature readings in the logs should match the ranges reported in Table 2 and Figure 14 (b) and Figure 15 (d)(e)(f) of the paper (e.g., Power $<$ 40W), substantiating the system's viability.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Revelio: A Global Study of VoIP Blocking

Arjun Gupta
IIT Bombay

Roya Ensafi
University of Michigan

Piyush Kumar
IIT Delhi

Devashish Gosain
IIT Bombay

A Artifact Appendix

A.1 Abstract

We present *Revelio*, a tool for detecting VoIP blocking at a global scale. *Revelio* uses interference in STUN traffic as a signal to detect VoIP blocking. *Revelio* supports scraping BGP prefix information, large-scale STUN probing using replayed VoIP App packets, STUN response deviation analysis, and traceroute-based middlebox inference.

Our artifact consists of two main components, (1) The codebase, which can be used for running the entire *Revelio* scans worldwide, and (2) The data we collected during our study, which can be used to reproduce the graphs from the paper.

A.2 Description & Requirements

Below, we detail the modules from both our codebase and data that are in accordance with the open science policy outlined in the accepted paper.

Codebase

- Scraping: it is used to web-scrape country ASNs and prefix data, merge and clean prefixes for further scanning.
- Zstun: perform protocol-level STUN probes using Zmap-assisted pipelines.
- Analysis & STUN Trace: post-process Zstun output and to produce per-country, analysis-ready graphs, filtered destination IPs and middleboxes.

Data

- Hyperquack data: remote measurement data for website blocking regarding VoIP apps (*graphs_from_paper/input/dataset_4*).
- Stun server IPs: censored STUN server IPs released after anonymizing the first two octets (*graphs_from_paper/input/dataset_i_anon*) where "i" can be 1, 2 or 3. This data is used to reproduce the graphs from the paper.

A.2.1 Security, privacy, and ethical concerns for the evaluators

Running *Revelio* poses no security, privacy, or ethical concerns for the evaluators in the uncensored region. However, if the evaluator is residing in a censored region (especially where VoIP is blocked), we recommend exercising caution when running the tool, as it involves sending potentially blocked STUN probes.

A.2.2 How to access

- The repository containing the code and data can be cloned from the following Github link.
- Archived versions of the artifact are available under the following Zenodo DOI.

A.2.3 Hardware dependencies

We recommend scanning machines provisioned with atleast 8 GB of RAM, 2 Intel vCPUs, and a 160 GB SSD, running Ubuntu 22.04 LTS (64-bit). Furthermore, since *Revelio* scans the IPv4 address space using `Zstun` (which is built on `Zmap` version 4.3.4), it requires a high network bandwidth for sending probes (we used 300 Mbps).

A.2.4 Software dependencies

We package all our dependencies as part of the code. Refer to section A.3 for detailed steps.

A.2.5 Benchmarks

None

A.3 Set-up

The repository provides a helper setup script (*setup.sh*) in the root directory to install the required dependencies and prepare the environment. At the end of execution, the script also verifies whether the installation was successful.

A.3.1 Installation

Run ***bash setup.sh*** to install the following dependencies and verify that each package has been installed successfully:

- Python dependencies
- Scapy and plotting libraries
- ZStun dependencies for compiling from source
- Required system packages
- Virtual environment setup components

A.3.2 Basic Test

There is a "Fast Mode" designed to perform an end-to-end run of *Revelio* with limited scope. To launch Fast Mode, please run ***bash run_fast.sh***.

As a representative example, the script only scans for the UAE IP space for WhatsApp VoIP call filtering. The output is a text file `filtered_ips_WhatsApp.txt` that contains the UAE IPs along which *Revelio* detected STUN filtering.

Next, the script attempts to estimate the location of the middleboxes. From this set of filtered IPs, the code randomly selects 10 IPs for which vanilla STUN probes yielded no response, but WhatsApp-specific STUN probes returned ICMP responses (see "No response censorship" in Section 5.3.2 of the paper).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): `Zstun` discovers usable STUN infrastructure that cannot be reliably identified using standard Internet-wide scanning approaches.

(C2): *Revelio* detects VoIP censorship through differences in responses for vanilla STUN, application-specific STUN, and modified application-specific STUN probes.

(C3): `StunTrace` can identify and characterize censorship middleboxes from filtered destination IPs and reproduce representative examples of the middlebox behaviors in a target country.

A.4.2 Experiments

(E1): *Zstun Discovery Evaluation [5 human-minutes + 15 compute-minutes (Demo Run) OR 21 compute-hour (Comprehensive Run)]* Linked to Claim C1.
Preparation: None
Execution: Navigate to the `claims/C1/` directory. Select either `run_demo.sh` to evaluate the claim on the United Arab Emirates or `run_comprehensive.sh` to evaluate the claim on all 19 countries reported in the paper.
Results: The scripts will result in three output text files corresponding to a target country. The files are `tcp_3478.txt` (contains the Zmap TCP output), `udp_3478.txt` (contains the raw Zmap UDP output), and `stun_a.txt` (contains the ZSTUN output). Finally, the script produces `claims/C1/c1_summary.csv`, which contains only the number of responsive IPs from each of the three files for easy comparison.
The reader can observe a typical trend: Number of UDP 3478 Open IPs < Number of STUN responsive IPs < Number of TCP 3478 Open IPs. This trend indicates that scanning UDP port 3478 discovers only a small fraction of usable STUN servers, whereas scanning TCP port 3478 identifies many hosts that do not actually provide STUN services. The key difference lies in the validation criterion: `Zstun` sends standard STUN probes and classifies an IP address as an operational STUN server only if it returns a valid STUN response. In contrast, ZMap's UDP/3478 and TCP/3478 scans merely indicate that the corresponding port is open and do not verify STUN functionality. Therefore, in our analysis, we treat `Zstun`-responsive IP addresses as operational STUN servers. By requiring successful STUN exchanges for identification, `Zstun` consistently discovers a stable set of genuine STUN servers, providing empirical support for the claim made in C1.

(E2): *VoIP Censorship Detection Evaluation [5 human-minutes + 18 compute-minutes]*. Linked to Claim C2.
Preparation: None
Execution: Navigate to the `claims/C2/` directory. Use `run_demo.sh` to evaluate the claim on the complete IP space of the target country. The country name and the app filtered in that country need to be passed as arguments to the script. By default, we selected the country "UAE" and the app "WhatsApp" whose STUN packets are subject to censorship in the UAE. Execute `run_comprehensive.sh` to evaluate the claim on all 19 countries reported in the paper. (Note that this should incur about 50 hours to complete the comprehensive scans from a single measurement machine.)
Results: For a target country, the `run.sh` code will generate VoIP censorship graphs. For example, if the target country is the UAE, the code will generate graphs similar to Figures 5 (UDP deviation), 7 (No response deviation), and 9–10 (ICMP deviation). Similar graphs will be created for different countries as an output after executing `run_comprehensive.sh`.
Interpretation of graphs: In the absence of filtering, we expect all three probe types (P1, P2, and P3) to show similar response behavior. However, for networks that perform VoIP filtering, P2 probes may exhibit noticeable deviations from P1 and P3 in UDP responses, ICMP errors, and non-responding IPs.

(E3): *Middlebox Localization and Characterization Evaluation [5 human-minutes + 30 compute-minutes]*. Linked to Claim C3.

Preparation: None

Execution: Navigate to the `claims/C3/` directory. Use `run.sh` to evaluate the claim on a representative set (10 IPs) of filtered destinations from each deviation category (UDP, ICMP, and No response) identified during the `Zstun` scan. The country name and the app filtered in that country need to be passed as arguments to the script. By default, we selected the country "UAE" and the app "WhatsApp" whose STUN packets are subject to censorship in the UAE. Execute `run_comprehensive.sh` to evaluate the claim on all 19 countries reported in the paper.

Results: The `run.sh` script will generate three output files for the specified target country, i.e., `<app_name>_middleboxes.txt`, `<app_name>_guessed_asn.txt`, and `<app_name>_likely_filtered_dest_IP.txt` for each deviation category. These files contain potential middlebox IPs (if visible in traces), potential ASes that perform filtering, and filtered destination IPs, respectively. In addition to these files, the vanilla and censored traces are stored in the trace_output directory. `run_comprehensive.sh` will generate such files, and the trace_output directory will be created for each of the 19 countries. (Note that this should incur about 30 hours to complete the comprehensive scans from a single measurement machine.)

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: PRIVACYSHIELD: Relaying BLE Beacons to Counter Unsolicited Tracking

Florian Hofhammer
EPFL

Daniele Antonioli
EURECOM

Mathias Payer
EPFL

A Artifact Appendix

A.1 Abstract

PRIVACYSHIELD is a relay-based system to protect users against unsolicited tracking via offline finding trackers such as Apple AirTags. These offline finding trackers emit Bluetooth Low Energy (BLE) beacons that are then picked up by nearby devices such as smartphones. These devices then upload their current location as an approximation of the trackers location to the vendor's servers, from which the owner of the tracker can then retrieve it.

As such trackers are small, easy to hide, and available at low cost, they are frequently abused for malicious purposes such as stalking. PRIVACYSHIELD aims to prevent such misuse by masking a potential victim's location. To this end, PRIVACYSHIELD relays AirTag beacons via a network of relay stations that reemit the beacons at different locations than where they were originally. Consequently, third-party devices detect the beacons at these locations and report the wrong location to the vendor's servers. The AirTag's real location is thus hidden from the owner of the tracker, which in this malicious case is the perpetrator.

This artifact appendix describes our artifact for PRIVACYSHIELD. The artifact contains the source code and documentation for *(i)* the relay application submitting AirTag beacons to the PRIVACYSHIELD server, *(ii)* the relay server collecting and redistributing AirTag beacons, *(iii)* the relay firmware reemitting beacons received from the server, and *(iv)* scripts to collect location records from Apple's servers for evaluation purposes.

A.2 Description & Requirements

This section describes the software and hardware requirements to reproduce a deployment of PRIVACYSHIELD.

A.2.1 Security, privacy, and ethical concerns

As described in the Ethical Considerations section of our paper, PRIVACYSHIELD selectively hides the location of trackers from their owners. While this is intended to protect potential victims from stalking, it may also temporarily and inadvertently prevent legitimate object tracking use cases. Therefore, we recommend evaluators to take precautonary measures when recording and relaying AirTag beacons, such as notifying people in the vicinity about the ongoing experiment and adjusting the minimal signal strength threshold in the AirGuard application to only capture beacons from AirTags in the direct vicinity, i.e., AirTags that are part of the experiment.

A.2.2 How to access

PRIVACYSHIELD is available at `https://doi.org/10.5281/zenodo.17964520` and `https://github.com/HexHive/privacyshield`. We use the (stable) Zenodo reference to point to the version of the artifact as it passed the Artifact Evaluation process. The (floating) GitHub reference points to the latest version of the artifact, which at the time of writing is identical to the Zenodo version and identified by the tag sec26-artifact-evaluated, commit 11a509cfcab20feaa12e735ef701366ba88f6066.

A.2.3 Hardware dependencies

The minimal hardware requirements for evaluating general functionality of PRIVACYSHIELD are as follows:

- an AirTag,
- an Android device with Bluetooth Low Energy (BLE) support to run the AirGuard application for recording and uploading beacons,
- an Apple device (iPhone, iPad, or Mac) to pick up the beacons and upload the location records to Apple's servers,
- a second Apple device to retrieve the location records and show the AirTag's location in the Find My application,
- a computer (physical or virtual) to run the PRIVACYSHIELD server (must be accessible from the Android device running the application as well as the relay station), and
- an ESP32-C3-based development board to act as a relay station. Other ESP32 variants may also work but have not been tested.

If one of the Apple devices used is a Mac, the PRIVACYSHIELD server can also be run on the same device. This does not accurately reflect a real-world deployment but is sufficient for the sake of the experiments.

Note that the above hardware requirements are minimal and can only verify the general functionality of PRIVACYSHIELD. In order to retrieve location reports from Apple's servers for evaluation purposes outside of the Find My application, a Mac is required as one of the Apple devices. This requirement stems from the need to extract the AirTag's private key from the macOS keychain, which is not possible on iOS or iPadOS devices.

A.2.4 Software dependencies

All the steps for building binary artifacts and running the relay server are containerized. The only software requirement is therefore a working Podman or Docker installation.

Note that the (optional) evaluation scripts based on Findmy.Py require a working Python3 installation and extracting AirTag's private keys from a macOS keychain first. This step is a prerequisite for the scripts and is not part of our artifact. We point the evaluator to upstream tutorials on how to perform this step, e.g., `https://docs.mikealmel.ooo/FindMy.py/getstarted/02-fetching.html`.

A.2.5 Benchmarks

None.

A.3 Set-up

Make sure to have the aforementioned software dependencies installed.

Furthermore, if you aim to retrieve location reports from Apple's servers outside of the Find My application, make sure to correctly extract the AirTag's private keys from a macOS keychain first. We refer to upstream tutorials such as `https://docs.mikealmel.ooo/FindMy.py/getstarted/02-fetching.html` for this step. This is not required for general functionality tests but was used to collect statistical information for our evaluation.

A.3.1 Installation

First, download the artifact, e.g., via `git clone https://github.com/HexHive/privacyshield`.

Set up and run the PRIVACYSHIELD server, the modified AirGuard application, and the relay firmware as described in the `README.md` in the repository. This includes installing required Python packages, applying our patches to upstream projects where necessary, and building and running binary artifacts and scripts.

A.3.2 Basic Test

If all the components are installed and running correctly, you can perform some basic tests to verify the general functionality of PRIVACYSHIELD.

To test whether the relay application is able to pick up AirTag beacons and upload them to the PRIVACYSHIELD server, start the application on the Android device and run `adb logcat --pid=$(adb shell ps | grep seemoo | cut -d ' ' -f 8) | grep insertScanResult` on a device that has `adb` access to the Android device, e.g., the machine you built the application on. Then, bring an AirTag in the vicinity of the Android device. If everything works correctly, you should see log messages indicating that AirTag beacons were picked up.

To test whether the server can receive beacons, run it with increased logging output, e.g., via `python3 server.py -v`. To test whether the server correctly redistributes beacons to relays, you can manually submit a beacon via `curl`: `curl -L --json '"data": "tyPkKWDIHv9MABIZEE2KEXyvZ+Vfq1kYAXoD35MRrlERI44CAA==", "valid_to": "2026-02-28 00:00:00"' "http://<server address here>/api/v1/airtag"`. This should successfully add the beacon to the server's database, which you can then retrieve via `curl -vL "http://<server address here>/api/v1/airtag?valid=true"`.

Once you configured and installed the relay firmware on the microcontroller development board, you can check its log outputs via a serial connection, e.g., using `screen /dev/ttyUSB0 115200` (assuming the serial device shows up as `/dev/ttyUSB0`, replace if necessary).

Optionally, as an end-to-end test, you can verify that relaying beacons works by bringing the AirTag close to the Android device running the application, and then removing the battery from the AirTag to stop it from emitting beacons. If everything works correctly, the AirTag should still show up as providing updates in the Find My application, as the relay station now effectively turns into a clone of the original AirTag.

A.4 Evaluation workflow

This section describes the steps required to evaluate the masking functionality of PRIVACYSHIELD based on a working system as set up in the previous section.

[Mandatory for Artifacts Functional & Results Reproduced, optional for Artifact Available] This section should include all the operational steps and experiments which must be performed to evaluate if your your artifact is functional and to validate your paper's key results and claims. For that purpose, we ask you to use the two following subsections and cross-reference the items therein as explained next.

A.4.1 Major Claims

(C1): PRIVACYSHIELD effectively masks the location of AirTags from their owner by relaying their beacons at a different location. We demonstrate this in the experiments described in Section 5.3 of the paper.

(C2): PRIVACYSHIELD can mask the location successfully even when there are significantly fewer devices picking up the relayed beacons compared to the original beacons. We demonstrate that this behavior can be achieved by adjusting the frequency at which the relays emit beacons. This is shown in the experiments described in Sections 5.4 and 5.5 of the paper.

A.4.2 Experiments

(E1): [Functionality Verification] [30 human-minutes]: This experiment verifies the functionality according to claim (C1).

Preparation: Make sure the Android application, the PRIVACYSHIELD server, and the relay firmware are correctly installed and running as described in Section A.3.

Execution: Bring the experiment AirTag close to the Android device running the sniffing application. With the relay station powered on and being located physically seperate from the AirTag, check whether the AirTag shows up in the Find My application at the relay stations position instead of its real position.

Results: The AirTag should show up in the Find My application at the relay station's position, demonstrating that PRIVACYSHIELD successfully masked the AirTag's real location.

(E2): [Verification of Resilience to Adversarial Environments] [30 human-minutes]: This experiment verifies that PRIVACYSHIELD can successfully mask the location of an AirTag even when there are significantly fewer devices picking up the relayed beacons compared to the original beacons, as described in claim (C2).

Preparation: Make sure the Android application, the PRIVACYSHIELD server, and the relay firmware are correctly installed and running as described in Section A.3. Ensure that the AirTag's beacons are picked up by multiple devices, e.g., by placing it in a busy area. Ensure that the relay station is placed in an area with significantly fewer devices picking up beacons, e.g., a controlled office environment.

Execution: Bring the experiment AirTag close to the Android device running the sniffing application. With the relay station powered on and being located physically seperate from the AirTag, check whether the AirTag shows up in the Find My application at the relay stations position instead of its real position. If this is not the case, adjust the frequency at which the relay emits the relayed beacons by reducing the `BLE_ADVERTISEMENT_INTERVAL` value for the relay firmware and repeat the experiment.

Results: The AirTag should show up in the Find My application at the relay station's position after adjusting the advertisement frequency. The results should mirror those presented in Table 2 in the paper, if the environment is similar to the one used in our evaluation.

Note that the above experiments are sufficient to validate the major claims of the paper. To get more fine-grained results that back up the claims made in the paper, the evaluator can optionally retrieve the history of location reports from Apple's servers using the scripts provided in the `findmy/` subdirectory of the repository. These scripts allow precomputing the AirTag's public keys for a configurable time frame, retrieving location reports, and plotting them on a map.

Warning

Interacting with Apple's servers with an unofficial client may get your account banned. If you choose to do those fully optional steps, make sure to use a secondary Apple ID!

A.5 Notes on Reusability

Our server implementation can easily be scaled horizontally by deploying it on multiple machines. These can then either serve disjoint sets of AirTag beacons, or share a common database by replacing the current SQLite backend with a networked database such as PostgreSQL, MySQL, or similar. Furthermore, the server can be federated by submitting beacons to other servers via the provided REST API.

The relay firmware can be adapted to other microcontroller platforms leveraging their built-in BLE capabilities and APIs. The current implementation is based on the Espressif SDK, and consequently should run successfully on any ESP32 variant that supports BLE and WiFi.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Do You Need a Receipt? Anonymous Credential Revocation at Continental Scale via Private Record Certification

Kasra EdalatNejad, Sebastian Faust, Jonas Hofmann, Philipp-Florens Lehwalder, Thomas Schneider
Technical University of Darmstadt, Germany
{edalat, schneider}@encrypto.cs.tu-darmstadt.de,
{sebastian.faust, jonas.hofmann1, philipp-florens.lehwalder}@tu-darmstadt.de

A Artifact Appendix

A.1 Abstract

We provide an artifact containing a proof-of-concept implementation of the Private Record Certification (PRC) protocol described in the paper. We also provide Docker files which automate the process of benchmarking our solution and generating the performance plots in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact is non-destructive and only collects performance measurement of its execution. There are no security, privacy, or ethical concerns.

A.2.2 How to access

The artifact is available on https://zenodo.org/records/20433787.

A.2.3 Hardware dependencies

This artifact requires an x86_64 CPU with AVX2 support. This includes most modern Intel and AMD CPUs, but excludes ARM-based platforms, including Apple Silicon.

Our benchmark evaluates up to 8 PRC servers. We measure single-core performance and restrict each server to one core. To simplify the process, we run all servers on the same machine with a simulated network. Therefore, we recommend running the full benchmark on a machine with at least 8 cores. If fewer cores are available, multiple servers will share cores, which may increase latency. The RAM footprint is low. A machine with 8 GiB of RAM should be sufficient for running the full 8-server benchmark.

A.2.4 Software dependencies

We provide a Docker container, based on Debian, to facilitate running the artifact. We require Docker to be installed on the device. We only provide instructions on how to run the container on a Linux machine, but it is possible to run it on different operating systems.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To install the artifact, download it or clone the corresponding Git repository. The artifact requires Docker; if Docker is not already installed, follow the official installation instructions.[1] In the artifact's top-level directory, build the container:

```
sudo docker build -t prc .
```

A.3.2 Basic Test

This artifact is built in Rust, and we use `cargo test` for basic tests. You can use the following command to run all tests in the container. We expect that all tests should pass.

```
sudo docker run \
 --rm \
 -v "$(pwd)/measurements:/AnonCred/measurements" \
 prc \
 cargo test
```

A.4 Evaluation workflow

A.4.1 Major Claims

This artifact reproduces the server-side PRC measurements (cf. Figure 3). The benchmark directly measures latency, com-

[1] https://docs.docker.com/get-docker/

putation cost, and communication cost and estimates the operation's monetary cost based on the AWS price model.

(C1): Sub-second latency at billion-credential scale. PRC achieves sub-second latency even for databases with over one billion credentials. The latency is dominated by the simulated network delay. We also note that when RTT is 20 ms, the latency is under 100 ms. This is proven by the latency measures provided in `latency.pdf` plot as part of experiment (E1).

(C2): Linear server computation. The computation cost of PRC servers grows linearly with the number of credentials. This is proven by the server computation cost plotted in `comp_comm.pdf` generated in experiment (E1).

(C3): Sublinear server communication. The communication cost of PRC servers grows sublinearly with the number of credentials; more precisely, it grows as $O(\sqrt{N})$. This is proven by server communication cost plotted in `comp_comm.pdf` generated in experiment (E1).

(C4): Efficient monetary cost at scale. The monetary cost grows linearly with the number of credentials, and the protocol remains efficient even when processing millions of PRC queries. The estimated monetary cost is computed from measured resource consumption using AWS pricing. Our pricing model is as follows: one CPU core, less than 2 GiB RAM, $0.04 per compute hour, free inbound transfer, and $0.09 per GiB outbound. Our cost efficiency is proven by the operation cost plotted in `costs.pdf` generated in experiment (E1).

The paper also makes the following client-side claims, which are not directly shown in the produced figures.

(C5): Client cost is independent of the number of revocations. In our protocol, revoking a credential only requires changing a value in the servers' database. There is no corresponding client-side update, and client queries are agnostic to the changes in the servers' database. The code shows that any change in revocation status is fully server-side, with no impact or notification to clients.

(C6): Client cost is logarithmic to the number of credentials. In theory, the client cost is logarithmic in the number of credentials because the record index size grows with the database size. In practice, we represent indices as a single integer where a `u32` supports up to four billion records. Therefore, within the evaluated range, the client cost is effectively independent of the number of credentials.

The paper compares the cost of supporting anonymous-credential revocation using our PRC construction against a naive PIR-based approach and ALLOSAUR. For the naive PIR baseline, the cost corresponds to performing a single PIR query over a database of n records, each of size 96 B. For ALLOSAUR, we use the implementation as provided, without modification.

Comparison with related work Because these comparisons evaluate unmodified external implementations—Hintless PIR[2], DPF-based PIR[3], and ALLOSAUR[4]—we do not include their source code in this artifact. A database with one billion 96 B entries requires 96 GB of storage, which is beyond the scalability of existing PIR implementations. We therefore extrapolate costs for settings in which prior systems fail to run.
We describe how we ran the external implementations, the raw measurements we obtained, and the extrapolation method used beyond their scalability limits. Since the external projects already provide their own installation and usage instructions, we do not include detailed setup instructions or automation for those packages.

A.4.2 Experiments

(E1): [human-interaction: minimal, compute-time: 3:30 min, peak memory: 1 GiB]: Run the automated evaluation.
How to: The evaluation is fully automated. After installing Docker and building the image per set-up instructions, run the following command:

```
sudo docker run \
 --cap-add=NET_ADMIN \
 --rm \
 -v "$(pwd)/measurements:/AnonCred/measurements" \
 prc \
 bash /AnonCred/bench_run_all.sh
```

Results: This script runs PRC protocols with $\{2,4,8\}$ PRC servers on a single machine with a simulated network—setting an RTT of $\{20\,\text{ms}, 40\,\text{ms}, 80\,\text{ms}\}$ via `tc` command—and measures the latency, server computation, and communication cost of running a PRC query based on the number of records held by the server. We run each experiment 5 times, record the performance measurement, and plot the costs. After the benchmark finishes, the following plots `latency.pdf`, `costs.pdf`, and `comp_comm.pdf` will be generated in the `measurements/` directory. These three plots correspond to Figure 3 in the paper and support our claims.

The PRC measurements reported in the paper are measure on a system running Debian 13 with an AMD Ryzen AI 7 PRO 350 CPU and 64 GB RAM.

A.5 Notes on Reusability

We provide detailed instructions on how to build the artifact from the source, manually run the server with customizable parameters, and describe the artifact structure and subcomponents that may be of independent interest, such as SIMD-aligned bit-arrays, our custom MPC engine, conversions, and PIR construction in the artifact readme. If you are interested in using/reusing our prototype, please refer to the latest version of the repository.

[2] https://github.com/google/hintless_pir
[3] https://github.com/google/distributed_point_functions
[4] https://github.com/sam-jaques/allosaurust

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Remise: Authorized Anonymous Communication Systems

Rohan Ravi, Partiosh Shukla, and Adithya Vadapalli

A Artifact Appendix

A.1 Abstract

This artifact provides the C++ prototype and benchmarking harness for REMISE, a two-server authorized anonymous communication system built on Distributed Oblivious RAM. It contains the two server-side benchmark binaries evaluated in the paper—`remise` (the REMISEBB bulletin-board variant) and `remise_sabre` (the REMISEBB (SABRE) variant whose write-authorization uses an MPC evaluation of a bitsliced LowMC PRF)—together with the DPF/FSS primitives, secret-sharing and MPC building blocks, and the coroutine-based asynchronous networking framework used for protocol execution. The harness runs the two servers as separate processes that communicate over a TCP socket, emulates wide-area network conditions with `tc netem` (latency) and `tbf` (bandwidth), and reports per-request audit latency, online validity-check latency, end-to-end throughput, and authorized-request goodput, appending each run to a CSV. We package everything as a two-container Docker setup (servers `p0` and `p1`) so that an evaluator can reproduce the request-validity-check and throughput/goodput trends with a small number of commands. The artifact reproduces the qualitative and quantitative trends of Figures 3 and 4 of the paper, including the order-of-magnitude online validity-check speedup of REMISEBB at large database sizes.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact runs entirely on the evaluator's own machine using *synthetic* data: databases, indices, messages, and authorization tokens are generated internally, and no real user data is collected, transmitted, or stored. There are no destructive operations on the host. Two points warrant the evaluator's attention:

- **Elevated container capabilities for network emulation.** To apply `tc netem`/`tbf` shaping *inside* the experiment containers, the two server containers are granted the Linux `NET_ADMIN` capability and are run with `apparmor=unconfined`. These settings are scoped to the two experiment containers on a private user-defined Docker bridge network; they do not alter the host's networking or security configuration. An evaluator who prefers not to grant these can run all experiments at zero emulated latency (LAN conditions) and still validate the computational trends—only the latency-dependent rows of Figure 3 require shaping.
- **Teardown.** The provided teardown script stops and removes only the two experiment containers and (optionally) the built image; it touches no other host state.

A.2.2 How to access

The artifact is archived on Zenodo at https://zenodo.org/records/20438384. Download and extract the archive; all paths below are relative to its root.

However, for the evaluation, we recommend cloning the repo from https://github.com/anonymoususer99946996/anonymous.git and then going into the Remise folder.

A.2.3 Hardware dependencies

- **CPU.** An x86-64 processor with the `AES-NI`, `SSE2`, `SSSE3`, and `AVX2` instruction-set extensions. `AES-NI` is used by the DPF pseudorandom generator; `AVX2` (256-bit vectors) is used by the bitsliced LowMC audit in REMISEBB (SABRE). The build uses `-march=native`, so the evaluation host should be the run host.
- **Memory.** Memory scales with database size and message size. The paper's measurements up to 2^{30} entries were taken on a machine with 251 GiB of RAM. The headline claims (database size 2^{26}) are reproducible with substantially less; we recommend at least $\approx$ 16–32 GiB to reach 2^{26} comfortably.
- **Cores / hosts.** The two servers may run as two containers on a single multi-core machine (the default), or on two separate machines on the same network. Two physical cores suffice; more cores do not change the reported per-request metrics.

The paper's reference platform was a server with two Intel® Xeon® Gold 6430 processors, 251 GiB of RAM, running 64-bit Linux.

A.2.4 Software dependencies

The recommended path is Docker (a single image builds both binaries). The provided `Dockerfile` is based on `ubuntu:22.04` and installs all dependencies; the only host requirement is a working Docker installation (Docker Engine with the `compose` plugin) on a Linux host.

For a native (non-Docker) build, the dependencies are: Ubuntu 22.04 (or equivalent), `g++-12` with C++20 support (required for coroutines), Boost 1.81 headers (the asynchronous networking layer uses `boost::asio` coroutines and `experimental/awaitable_operators`, which are not present in Ubuntu's default Boost 1.74), `libbsd` (`arc4random`), OpenSSL/`libcrypto` (used by the LowMC audit streams), `iproute2` (`tc`, for latency/bandwidth emulation), and `make`. The Boost 1.81 headers are vendored into the image at build time, so the Docker path needs no separate Boost installation.

A.2.5 Benchmarks

None. All workloads (databases, write/read requests, authorized vs. cover traffic, authorization tokens) are generated internally and parameterized on the command line. No external data sets or models are required.

A.3 Set-up

A.3.1 Installation

1. Install Docker Engine with the `compose` plugin on a Linux host.
2. From the artifact root, build the image and start the two server containers:
   ```
   ./up.sh            # build & start
   ./up.sh --rebuild  # force rebuild
   ```
   The build compiles both benchmark binaries (`make remise` and `make remise_sabre`) and verifies that `/app/remise` and `/app/remise_sabre` exist in the image. The two containers (`p0`, the server/acceptor, and `p1`, the client) then run idle on a private bridge network, ready to receive experiments. Results are written to the host directory `./results/`, which is mounted into the containers.
3. Tear everything down when finished:
   ```
   ./down.sh          # stop & remove
   ./down.sh --rmi    # and remove image
   ```

Native build (optional). With the dependencies above installed and the Boost 1.81 headers available, run `make remise` and `make remise_sabre` (the Makefile honors `BOOST_ROOT=$HOME/boost-1.81`).

A.3.2 Basic Test

With the containers up, run a single small REMISEBB (SABRE) request at 10 ms RTT and 100 Mbit/s, a 2^{16}-entry database, 32-byte messages (leaf size 2), all requests authorized, 4 requests:

```
./run_remise_sabre.sh 10 100mbit 16 2 1.0 4
```

The script shapes the link, launches `p0` (server, backgrounded) and then `p1` (client, foreground), waits for completion, and clears the shaping. Expected output ends with an `Experiment Summary` block reporting (among others) the average audit latency, the average online latency (audit → finalize → FCW exchange → database update), bandwidth, throughput, and goodput, and appends a row to `./results/`. An analogous smoke test for REMISEBB uses `./run.sh 10 100mbit 16 2 1.0 4`. Successful completion of either (a printed summary and a new CSV row, no crash) confirms the build, the two-party networking, and the latency shaping are all functioning.

Execution. The experiments are driven by per-figure sweep scripts that loop over the database sizes, latencies, and authorized fractions of the corresponding figure and append one CSV row per trial; companion Python scripts read those CSVs and emit the plots. The underlying run scripts are:

`run.sh` one REMISEBB run:

```
./run.sh <RTT_ms> [bw] [log_nitems]
[leafsize] [auth_fraction] [num_requests]
[batch_size]
```

`run_remise_sabre.sh` one REMISEBB (SABRE) run:

```
./run_remise_sabre.sh <RTT_ms> [bw]
[log_nitems] [leafsize] [auth_fraction]
[num_requests]
```

where `log_nitems` is $\log_2$ of the database size, the message size in bytes is $16 \times$ `leafsize` (so `leafsize` $= 2$ gives the 32-byte messages used throughout), and `auth_fraction` is the *fraction of authorized requests*—i.e. the paper's "f% unauthorized traffic" corresponds to `auth_fraction` $= 1 - f/100$. The run scripts forward the requested RTT and a trial index to the binary (via the `REMISE_RTT_MS` and `REMISE_TRIAL` environment variables) so that each CSV row is self-labeled with its latency and repetition.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *The server-side request-validity-check (audit + authorization) time of* REMISEBB *is competitive at small*

database sizes and increasingly favorable at large ones; considering only the online phase, REMISEBB *attains an order-of-magnitude speedup over Spectrum (PACL), reaching nearly* $80\times$ *at database size* 2^{26}. This is evaluated by experiment (E1) and corresponds to Figure 3a and the discussion in Section 5 ("Request Validity Check Comparisons").

(C2): *The request-validity-check time of* REMISEBB (SABRE) *is essentially independent of the database size, because its PRF-based authorization is* $O(1)$ *and its audit is* $O(\log n)$*; consequently it is increasingly favorable relative to Express (PACL), whose DPF-based authorization grows linearly with the database size.* This is evaluated by experiment (E2) and corresponds to Figure 3b.

(C3): *For both* REMISEBB *and* REMISEBB (SABRE)*, throughput and goodput behave as reported in Figure 4: with offline-preprocessed DPFs, increasing the fraction of unauthorized traffic* improves *throughput (unauthorized requests terminate after the lightweight authorization phase and skip the expensive access phase), whereas generating DPFs on-the-fly reduces both; and* REMISEBB (SABRE) *with on-the-fly DPFs surpasses Express (PACL) at progressively smaller database sizes as the unauthorized fraction grows.* This is evaluated by experiment (E3) and corresponds to Figures 4a and 4b.

(C4): *For both* REMISEBB *and* REMISEBB (SABRE)*, the preprocessing (DPF interval evaluation) time and the audit time are independent of the message size, whereas the access time (finalize, FCW exchange, and database update) grows linearly with the message size.* This is evaluated by experiment (E4). This claim corresponds to Figure 5 from our paper.

Scope note on baselines. Reproducing the Spectrum (PACL) and Express (PACL) curves requires the PACL codebase (https://github.com/sachaservan/pacl; they are also available in the `pacl` folder in our artifact); they are currently not part of this artifact. For the *Functional* and *Results Reproduced* badges, the experiments below focus on reproducing the REMISE curves of Figures 3 and 4 (the contribution of this paper); the cross-system speedup factors in C1–C3 are then obtained by comparing against the published PACL measurements (or against a local PACL build, if the evaluator chooses to set one up).

A.4.2 Experiments

Timing conventions used below. Each request proceeds through *eval* (the `__evalinterval_mpc` DPF interval evaluation), *audit* (the validity check—an XOR-tag exchange for REMISEBB, a bitsliced-LowMC MPC for REMISEBB (SABRE)), and *write* (finalize, the FCW exchange, and the database update). DPF *generation* is offline and is never inside any timed region. We report two timings that differ only in whether *eval* is counted:

- **online** = audit + finalize + FCW exchange + DB update (eval *excluded*); this is the "with preprocessed DPFs" setting.
- **total** (a.k.a. "on-the-fly") = eval + audit + write (eval *included*, generation still excluded). This called "on-the-fly", because here we assume that the DPFs are produced as and when we need it.

Both numbers are produced by a *single* run—no separate mode flag is needed—because they are simply the same work measured against two different denominators.

(E1): *RemiseBB validity-check time (Figure 3a)* [15 human-minutes + ≈2–4 compute-hours + a few GiB to ∼tens of GiB RAM depending on the largest database size]: measure the per-request validity-check time of REMISEBB as a function of database size, at three latencies, reporting both the *online* time (audit only, eval excluded) and the *total* time (eval + audit), underlying C1.

Preparation: Bring the containers up (`./up.sh`). The sweep uses 32-byte messages (`leafsize` = 2), all-authorized traffic (`auth_fraction` = 1.0), exactly one request per run (`num_requests` = 1, `batch_size` = 1), and averages over 30 trials per point. The default database range is 2^{16} through 2^{26} at latencies 10/30/60 ms; pass arguments to cap the size or trial count (e.g. `./sweep_fig3a.sh 18 3` for a quick check).

Execution: Collect the data and produce the plots:

```
./sweep_fig3a.sh
python3 plot_fig3a.py --combined
```

Results: `sweep_fig3a.sh` writes one row per trial to `results/fig3a_remisebb.csv` with columns `variant,latency_ms,log_nitems,trial,online_ms,total_ms`, where `online_ms` is the audit-only time and `total_ms` is eval+audit. `plot_fig3a.py` averages over the 30 trials (with 95% confidence intervals) and emits one plot per latency (10/30/60 ms) plus a combined panel, each showing the *online* and *total* curves vs. database size. Compare against Figure 3a. To obtain the C1 speedup factor, divide the Spectrum (PACL) online time (from the paper, or a local PACL run) by the REMISEBB `online_ms` at the same database size; at 2^{26} this ratio should be close to $80\times$.

(E2): *RemiseBB (Sabre) audit time (Figure 3b)* [15 human-minutes + ≈1–2 compute-hours]: measure the audit (validity-check) time of REMISEBB (SABRE) as a function of database size and confirm it is (near-)constant in the database size, underlying C2. The Sabre audit is the bitsliced-LowMC MPC (`encrypt2_p0p1`); there is no preprocessing split, so a single *audit* time is reported.

Preparation: Same workload as E1 (`leafsize` = 2, `auth_fraction` = 1.0, one request per run, 30 trials, 2^{16} through 2^{26} at 10/30/60 ms), but driven through `remise_sabre`.

Execution: `./sweep_fig3b.sh`
`python3 plot_fig3b.py --combined`
Results: `sweep_fig3b.sh` writes one row per trial to `results/fig3b_remisebb_sabre.csv` with columns `variant,latency_ms,log_nitems,trial,audit_ms`. `plot_fig3b.py` averages over trials and plots audit time vs. database size for each latency; confirm the curve stays essentially flat as the database grows (the audit cost is independent of the database size), in contrast to Express (PACL) whose authorization grows linearly. Compare against Figure 3b.

(E3): *Throughput and goodput, with-preproc and on-the-fly DPFs (Figures 4a and 4b)* [20 human-minutes + ≈2–3 compute-hours]: measure throughput and goodput while varying database size and the fraction of unauthorized traffic, for both REMISEBB (4a) and REMISEBB (SABRE) (4b), underlying C3.
Preparation: The emulated latency is fixed at 30 ms and the message size at 32 bytes (`leafsize` $= 2$). The figure's three panels correspond to 30%/60%/90% unauthorized traffic, i.e. `auth_fraction` $\in \{0.7, 0.4, 0.1\}$; the sweep iterates these automatically. Throughput is meaningful only with many requests per run, so the sweep uses a large `num_requests` (default 200) and several trials. *No DPF-mode flag is required:* each run reports both the with-preproc curves (*online* throughput/goodput, eval excluded) and the on-the-fly curves (*total* throughput/goodput, eval included).
Execution: Collect both variants and plot each figure:

```
./sweep_fig4.sh both
python3 plot_fig4.py \
    results/fig4a_remisebb.csv \
    --out fig4a \
    --title "RemiseBB"
python3 plot_fig4.py \
    results/fig4b_remisebb_sabre.csv \
    --out fig4b \
    --title "RemiseBB (Sabre)"
```

(Use `./sweep_fig4.sh a` or `b` to run a single variant, and a trailing `MAXK TRIALS REQS` to scale down for a quick check, e.g. `./sweep_fig4.sh a 18 2 50`.)
Results: `sweep_fig4.sh` writes one row per trial to `results/fig4a_remisebb.csv` and `results/fig4b_remisebb_sabre.csv` with columns `variant,unauth_frac,log_nitems,trial,throughput,goodput,online_throughput,online_goodput`. Each `plot_fig4.py` call averages over trials and renders the three-panel row (30%/60%/90% unauthorized), with four curves per panel: with-preproc throughput/goodput (the `online_*` columns) and on-the-fly throughput/goodput (the plain columns). Confirm the three qualitative effects of C3: (i) with preprocessed DPFs, higher unauthorized fraction raises throughput/goodput; (ii) on-the-fly DPFs lower both; and (iii) for REMISEBB (SABRE), the on-the-fly curves overtake Express (PACL) at smaller database sizes as the unauthorized fraction grows. Compare against Figures 4a and 4b.

(E4): *Message-size scaling of preprocess, audit, and access* [15 human-minutes + ≈1 compute-hour]: measure how the three per-request cost components scale with message size, for both REMISEBB and REMISEBB (SABRE), underlying C4. The three components are *preprocess* (the `__evalinterval_mpc` DPF interval evaluation), *audit* (the validity check), and *access* (finalize, FCW exchange, and database update).
Preparation: The database size is fixed at 2^{20} and the latency at 30 ms; the message size is swept by varying `leafsize` $\in \{2, 64, 256\}$ (i.e. 32, 1024, and 4096 bytes, since message size $= 16 \times$ `leafsize`). Each point uses one authorized request per run and averages over 30 trials. The sweep runs both binaries.
Execution: Collect both variants and plot each:

```
./sweep_msgsize.sh both
python3 plot_msgsize.py \
    results/msgsweep_remisebb.csv \
    --out msg_remisebb \
    --title "RemiseBB"
S=results/msgsweep_remisebb_sabre.csv
python3 plot_msgsize.py "$S" \
    --out msg_sabre \
    --title "RemiseBB (Sabre)"
```

Results: `sweep_msgsize.sh` writes one row per trial to `results/msgsweep_remisebb.csv` and `results/msgsweep_remisebb_sabre.csv` with columns `variant,latency_ms,leafsize,log_nitems,trial,preprocess_ms,audit_ms,access_ms`. Each `plot_msgsize.py` call averages over the 30 trials (with 95% confidence intervals), prints a summary table, and renders a stacked bar chart of preprocess/audit/access vs. message size. Confirm C4: the preprocess and audit segments stay essentially constant across the three message sizes, while the access segment grows roughly linearly with the message size.

In all experiments the per-trial CSV files (and the plots derived from them) are the primary artifacts of record; the network conditions are those reported in the paper (`tc netem` latency, bandwidth shaping via `tbf`). Expected outcomes are the curves of Figures 3 and 4; small deviations in absolute timing are expected across hardware, but the trends—and the order-of-magnitude online speedup underlying C1—should hold.

A.5 Notes on Reusability

The harness is parameterized so evaluators can explore beyond the reported points. `log_nitems` scales the database

size; `leafsize` scales the message size (bytes $= 16 \times$ `leafsize`); `auth_fraction` sweeps the authorized/cover-traffic mix; `num_requests` sets how many requests a run issues; and (for REMISEBB) `batch_size` controls how many requests are pipelined concurrently. Network conditions are independent of the binaries: `netshape.sh` applies and clears latency (`netem`) and rate (`tbf`) shaping inside the containers, splitting the requested RTT across the two endpoints, so arbitrary latency/bandwidth points can be evaluated without rebuilding. Because the two servers communicate only over a TCP socket and `p1` resolves its peer via the `P0_HOST` environment variable, the same binaries can be run across two physical machines (set `P0_HOST` to `p0`'s address and open port `9200`) to measure over a real network instead of an emulated one. The per-figure sweep scripts (`sweep_fig3a.sh`, `sweep_fig3b.sh`, `sweep_fig4.sh`) and their plotting companions (`plot_fig3a.py`, `plot_fig3b.py`, `plot_fig4.py`) are thin wrappers over this interface and are easy to adapt to new latency points, database sizes, or authorization mixes.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: SophOMR: Improved Oblivious Message Retrieval from SIMD-Aware Homomorphic Compression

Keewoo Lee
Ethereum Foundation

Yongdong Yeo
Seoul National University

A Artifact Appendix

A.1 Abstract

We provide a C++ reference implementation of the Oblivious Message Retrieval (OMR) and Oblivious Message Detection (OMD) schemes, referred to as SophOMR and SophOMD, respectively, in the accompanying paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact does not pose any security, privacy, or physical risks to evaluators, nor does it raise ethical concerns. The payloads used to simulate data posted on a bulletin board are randomly generated.

A.2.2 How to access

The artifact is available at: https://doi.org/10.5281/zenodo.17958466.

A.2.3 Hardware dependencies

The artifact has no specific hardware dependencies and can be run on standard platforms. However, the HEXL backend is optimized for processors with AVX512_IFMA support (e.g., Intel Ice Lake). The benchmarks reported in our paper were run single-threaded on a Google Compute Cloud `n2-standard-8` instance (Intel Ice Lake, 2.6 GHz CPU with 32 GB RAM[1]).

A.2.4 Software dependencies

The artifact is based on the following dependencies:

- C++ build environment
- CMake build infrastructure
- NTL library
- OpenFHE library (Tested with v1.4.0[2])
- HEXL library (Optional; Skip on ARM)

The benchmarks reported in our paper were run on Ubuntu 20.04 LTS with OpenFHE v1.2.0.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

The commands below are based on Ubuntu 24.04 LTS with x86-64 and assume the use of the HEXL backend.

1. Install CMake, GMP, and NTL (if needed).

```
sudo apt-get update
sudo apt-get install build-essential
sudo apt-get install cmake
sudo apt-get install libgmp3-dev
sudo apt-get install libntl-dev
```

2. Install OpenFHE + HEXL. Follow the instructions at https://github.com/openfheorg/openfhe-hexl, or run:

```
git clone https://github.com/openfheorg/
    openfhe-configurator.git
cd openfhe-configurator
# To reproduce the version used in the paper, run:
    git reset --hard 9ac9090
scripts/configure.sh

# Would you like to stage an openfhe
    -development build?      [y/n] : n  (y on ARM)
# Would you like to stage an openfhe-hexl
    build?                   [y/n] : y

sudo scripts/build-openfhe-development.sh
```

[1] The installation of OpenFHE and HEXL may require more than 32 GB of RAM (we were able to install using 64 GB RAM). After installation, however, SophOMR runs comfortably with 32 GB of RAM.

[2] To implement ring-switching, we use OpenFHE in a manner not officially supported by its APIs, which may be incompatible with OpenFHE versions beyond 1.4.0.

3. Build the Library.

```
cd ..  # if still in openfhe-configurator
    directory
git clone https://github.com/keewoolee/
    SophOMR.git  # clone the repository
cd SophOMR
mkdir build
cd build
cmake .. -DCMAKE_PREFIX_PATH=~/openfhe
    -configurator/openfhe-staging/install
    # adjust the path to the location of the
    openfhe libraries
make
```

A.3.2 Basic Test

The following command runs the fastest predefined configuration (1-2 compute-minutes). On success, the final line of the output will be: `Result is Correct!`

```
./test OMD 65536 16
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *SophOMR achieves, compared to the prior state of the art PerfOMR, reductions of* 3.4× *in runtime,* 2.2× *in digest size, and* 1.5× *in key size in a setting with 65,536 payloads (each 612 bytes), of which up to 50 are pertinent. In particular, the runtime of the homomorphic compression step is reduced by* 7.5×*. These results are reported in Tab. 4 and Fig. 2, together with the corresponding OMD variants (SophOMD and PerfOMD). Similar trends are observed for 524,288 payloads (Tab. 8 and Fig. 4). The SophOMR and SophOMD runtimes presented in the tables and figures are reproducible using Experiment E1.*[3]

(C2): *SophOMR and SophOMD exhibit trends consistent with Fig. 3 of our paper for varying values of k and t. The SophOMR and SophOMD runtimes presented in Fig. 3 are reproducible using Experiment E1.*

A.4.2 Experiments

(E1): *Full Benchmarks (6-7 compute-hours): This experiment runs all benchmarks presented in the paper and supports Claims C1 and C2. Each benchmark is repeated five times, and we report the average runtime along with the standard deviation.*
Execution: *In the build directory, run the following command:*

```
python3 -u ../benchmark.py > benchmark.txt 2>&1
```

Results: *The file* `benchmark.txt` *contains the timing results and corresponding standard deviations for all benchmarks presented in the paper. The results are organized by the figures and tables in which they appear. When run in the same environment described in Section A.2.3 and A.2.4, the output should reproduce the reported results; in other environments, it is expected to follow similar performance trends.*

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/usenixsec2026/.

References

[1] Zeyu Liu, Eran Tromer, and Yunhao Wang. PerfOMR: proof of concept C++ implementation for OMR (Oblivious Message Retrieval) with Reduced Communication and Computation. https://github.com/ObliviousMessageRetrieval/ObliviousMessageRetrieval/tree/perfomr.

[3] PerfOMR is outside the scope of this artifact. The runtime results reported in our paper can be obtained using the original PerfOMR implementation [1]. To ensure reproducibility, we provide a fork of the original repository with minor modifications necessary for our evaluation, which reproduces the results by following the provided instructions. The fork is available at: https://github.com/keewoolee/PerfOMR.



USENIX Security '26 Artifact Appendix: SIGMA-RS: A Modular Approach for Keyed-Verification Anonymous Credentials

Michele Orrù
CNRS

Lindsey Tulloch
Tor Project, University of Waterloo

Victor Snyder-Graf
RISC Zero

Ian Goldberg
University of Waterloo

A Artifact Appendix

A.1 Abstract

We introduce a new software stack in Rust aimed at simplifying constructions and deployments of protocols based on modern anonymous credential systems.

The stack, called SIGMA-RS, through its layered design, abstracts cryptographic complexity while remaining flexible enough to support a range of credential schemes, proofs, and access policies. It emphasizes misuse resistance via type safety, domain separation, and prover-state discipline, and supports side-channel-aware constant-time strategies.

We evaluate practicality through re-implementations of Tor's Lox bridge distribution protocols and of user authentication in the Open Observatory for Network Interference (OONI).

The artifact will allow you to run all of the self-tests in our SIGMA-RS stack and the Lox and OONI applications, and to reproduce Tables 2, 3, and 5 in the paper.

A.2 Description & Requirements

This repository contains the code artifact for SIGMA-RS, a Rust software stack for implementing protocols based on keyed-verification anonymous credentials (KVAC).

The SIGMA-RS stack consists of the following components (listed along with the versions pinned in this artifact):

- spongefish tag v0.6.1
- sigma-proofs tag v0.3.1
- sigma-compiler tag 0.2.2
- cmz tag 0.2.1

This artifact also evaluates two sample applications using this stack, described in the paper:

- application-ooni tag artifact-v0.4
- application-lox tag lox-artifact

We also include for comparison a version of application-lox that uses an update of the older zkp crate, instead of our SIGMA-RS stack:

- application-lox-zkp tag lox-artifact-zkp

Artifact structure. The directories in this repository are as follows:

Scripts: Useful scripts for building a docker image for this artifact, and running tests therein

patches: Patches to the Cargo.toml file for our SIGMA-RS collection of crates, to force them to use the local copies of each other as dependencies, rather than using the published versions from crates.io. These patches are applied automatically by the `build-docker` script.

A.2.1 Security, privacy, and ethical concerns

There is no risk to the evaluators.

A.2.2 How to access

Our artifact is available at `https://git-crysp.uwaterloo.ca/SigmaProtocol/sigma-rs-artifact` with an archival copy at `https://doi.org/10.5281/zenodo.20190651`.

A.2.3 Hardware dependencies

Most of the results in this artifact can be reproduced on any machine that can run a recent Linux distribution (64-bit x86 or ARM). For best results (more closely matching those in the paper), use a CPU with AVX512 support. However, the artifact will still work, albeit with slower results, if you have only AVX2 or no AVX support at all.

To execute the OONI iOS benchmarks (the "iOS" column in Figure 3 in the paper), you will need a Mac host with Xcode and rustup installed, and an iOS device on which you can install apps you compile yourself.

A.2.4 Software dependencies

We assume your host runs Ubuntu 24.04, but any similar system should be fine. You will need installed on the host:

- git
- Either docker or podman. The scripts will auto-detect which of docker or podman you have. The *names* of the scripts contain the word "docker", even if they end up using podman instead.
- A wasm-capable web browser, which is pretty much any modern browser. You cannot use Tor browser, since you'll be connecting to a localhost web server, which you cannot do over the Tor network.
- As noted in Section A.2.3, the Mac host used to execute the OONI iOS benchmarks will need to have Xcode and rustup installed.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

After downloading or cloning the repository, build a docker image with:

```
./Scripts/build-docker
```

On a recentish laptop, this image should take around 10 minutes to build. The resulting image is about 9 GB.

A.3.2 Basic Test

To ensure everything has built properly, you should run the unit tests within the docker with:

```
Scripts/run-docker Scripts/run_all_tests
```

This should take less than 30 seconds to run.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The timings of our modular SIGMA-RS re-implementation of the Lox protocols are reasonable for modern web applications such as Tor for each of the protocols in both the native and wasm environments. They are also comparable to those of the original Lox implementation using the zkp library. This claim is supported by Experiment (E1), whose results are reported in Section 8.1 of the paper (Table 2).

(C2): The request and response sizes of our modular SIGMA-RS re-implementation of the Lox protocols are comparable to those of the original Lox implementation using the zkp library. This claim is supported by Experiment (E1), whose results are reported in Appendix D of the paper (Table 5).

(C3): Our implementation of the OONI protocols using our SIGMA-RS stack shows protocol operations completing within a few milliseconds on both the client and server, with comparable performance across the native and iOS platforms and no significant overhead introduced by mobile execution. This claim is supported by Experiments (E2) and (E3), whose results are reported in Section 8.2 of the paper (Table 3).

A.4.2 Experiments

(E1): *[Lox native and wasm benchmarks] [a few minutes]: This experiment runs both the original implementation of the Lox protocols using the zkp library, as well as our re-implementation using our* SIGMA-RS *stack. Both implementations are run in both native code and wasm environments. The time to perform the protocols and the sizes of the request and response messages are reported. This experiment supports Claims (C1) and (C2).*

Preparation: Have built the docker (or podman) image following Section A.3.1 above. Open a wasm-capable browser (running on the same machine that is running the experiment); be ready to load a URL with it. See Section A.2.4 above for more information about the wasm-capable browser.

Execution: To run the Lox native and wasm benchmarks (Tables 2 and 5 of the paper):

```
Scripts/run-docker Scripts/run_lox_benches
```

When the wasm benchmarks are run (twice: once for the new SIGMA-RS version of Lox, and once for the original zkp version), the script will prompt you with a URL to load in the wasm-capable web browser. Do so when prompted each of the two times.

Results: The output of the script will end with the data tables corresponding to Tables 2 and 5 in the paper. The values in Table 2 are times in milliseconds (with stddevs in parens). The values in Table 5 are sizes in bytes.

(E2): *[OONI native benchmarks] [30 seconds]: This experiment runs our* SIGMA-RS *implementation of the OONI protocols in the native environment. The times to perform the protocols are reported. This experiment supports Claim (C3).*

Preparation: Have built the docker (or podman) image following Section A.3.1 above.

Execution: To run the OONI native benchmarks (the "native" columns in Table 3):

```
Scripts/run-docker Scripts/run_ooni_benches
```

Results: The output of the script will be the data tables corresponding to the "native" columns of Table 3 in the paper. The values are times in milliseconds (means and stddevs).

(E3): *[OONI iOS benchmarks] [2 minutes combined for preparation and execution]: This experiment runs our* SIGMA-RS *implementation of the OONI protocols in the iOS environment. The times to perform the protocols are reported. This experiment supports Claim (C3).*

Preparation: As mentioned in Section A.2.3 above, this experiment must be run on a Mac host with Xcode and rustup installed, and an iOS device on which you can install apps you compile yourself.

If you downloaded the artifact from zenodo, or if you built the docker image following Section A.3.1 above, then you will have an `application-ooni` directory. Otherwise (for example, if you do not have docker on your Mac host, and you cloned the artifact repository from git-crysp.uwaterloo.ca), you can clone the dependency repositories with:

```
Scripts/clone-repos
```

In either case, once you have the `application-ooni` directory:

```
cd application-ooni
rustup target add aarch64-apple-ios \
   aarch64-apple-ios-sim x86_64-apple-ios
./ios/build-ffi.sh
```

This writes the file `ios/OoniAuthBindings/OoniAuthFFI.xcframework`.

Open `ios/OoniAuthApp.xcodeproj` in Xcode, select the `OoniAuthApp` target, build and then run, selecting your iOS device as the destination.

If the build fails with `xcodebuild ... requires Xcode` or a missing iOS SDK:

```
sudo xcode-select -s \
 /Applications/Xcode.app/Contents/Developer
```

Execution: Open the app, which has a single button. Press the button.

Results: The output of the app will be the data corresponding to the "iOS" column of Table 3 in the paper. The values are times in milliseconds (means and stddevs).

A.5 Notes on Reusability

For instructions on using the SIGMA-RS stack to implement your own KVAC protocols, see README.md in the cmz 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: FIRA: Enabling Automatic Forensic Investigation of Unmanned Aerial Vehicles

Yizhi Huang, David Oygenblik, Runze Zhang, Mingxuan Yao, Muhammad Ibrahim,
Burak Sahin, Haichuan Xu, Saman Zonouz, Brendan Saltaformaggio
Georgia Institute of Technology

A Artifact Appendix

A.1 Abstract

This artifact accompanies the FIRA paper and provides the complete pipeline for reproducing the forensic investigation and reconstruction of autonomous drone accidents. The artifact includes: (1) firmware causal graph extraction and validation for PX4 and autopilot, (2) neural network causality analysis via layer conductance attribution, (3) model reconstruction from mission-collected neuron data, and (4) evaluation of poisoned vs. clean models before and after online learning updates. All experiments, data, and pre-trained models are included and can be executed end-to-end via a single Python script or Docker container.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact includes pre-trained ML models (some intentionally poisoned for evaluation purposes) and drone mission telemetry data. All data collected from controlled test environments. No real accident data is included. The poisoned models are used only for demonstrate forensic capabilities and have no security risk to the evaluator's machine. The Docker container runs with default permissions and does not require elevated privileges beyond Docker access.

A.2.2 How to access

The archived artifact is available on Zenodo at: https://zenodo.org/records/17945698

Evaluators can also access a pre-configured evaluation machine via SSH:

```
ssh -i /path/to/key -p 10467
usenix_ae@0.tcp.ngrok.io
```

The SSH private key is provided separately on the HotCRP website.

A.2.3 Hardware dependencies

The experiments require a CUDA-capable GPU for neural network inference and attribution analysis. Our evaluation environment uses:

- 1× NVIDIA A6000 GPU (48 GB VRAM)
- 128 GB system RAM
- about 20 GB disk space

A pre-configured machine with the above hardware is accessible via the SSH.

A.2.4 Software dependencies

The artifact is packaged as a Docker container. The only host requirement is:

- Docker Engine with NVIDIA Container Toolkit

Alternatively, for running without Docker, the following are required:

- Ubuntu 20.04+ (or equivalent Linux distribution)
- Miniconda / Anaconda with Python 3.8
- PyTorch 2.x with CUDA 12.1 support
- Python packages: `matplotlib`, `captum`, `trojai`, `Levenshtein`, `networkx`, `tqdm`, `pandas`, `numpy`, `Pillow`

A.2.5 Benchmarks

All required data is included in the artifact:

- **Clean test images**: 1000+ drone camera images for autonomous landing classification (10 classes), located in `AE/clean_data/`.
- **Poisoned test images**: Adversarial variants of the test images, located in `AE/poison_data/`.

- **Pre-trained models**: Baseline MobileNetV2 model for autonomous landing (`pre_mission/model/auto_land/auto_land.pth`) and a poisoned variant (`during_mission/online_learning_model/auto_land/poison.pth`).
- **Mission-collected neuron data**: Pickle files containing critical neuron weights captured during simulated missions (`during_mission/mission_collected_data/auto_land/`).
- **Firmware causal graphs**: PX4 v5x extracted module causality data (`AE/v5x/BCG/`).

A.3 Set-up

A.3.1 Installation

Option A: Docker. Build and run the Docker container from the artifact root:

```
docker build -t fira_ae .
docker run --gpus all -it fira_ae
```

Option B: Manual installation. From the artifact root directory:

```
conda create -n FuDrone python=3.8 -y
conda activate FuDrone
pip install torch torchvision torchaudio \
--index-url https://download.pytorch.org/whl/cu121
pip install matplotlib captum trojai Levenshtein \
networkx --no-deps
pip install matplotlib captum Levenshtein networkx
```

A.3.2 Basic Test

Activate the environment and run the first experiment as a test:

```
conda activate FIRA
cd FIRA_AE/AE/v5x/BCG
python before_check.py
```

Expected output: the script prints the number of nodes in the system causal graph, the number of matched nodes in the ground truth, and an ED score, in Table 3.
e.g.:

```
Nodes in SYS: <N>
Matched in GT: <M>
ED: 0.XXX
```

If the script completes without errors and prints an ED score, the environment is correctly configured.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): FIRA can extract and validate firmware causal graphs from drone binaries, and the pruned causal graph retains high equivalence to the ground truth. This is supported by experiment (E1) (Table 3, rows 1-3 in the original paper).

(C2): FIRA can identify critical neuron groups in neural network models using layer conductance attribution, producing model causal graphs (MCGs) that accurately characterize model behavior. This is supported by experiment (E2) (Table 2, rows 1-3 in the original paper).

(C3): FIRA can reconstruct neural network models from collected neuron data transmitted during flight, and the reconstructed models have distinguishable accuracy differences between clean and poisoned inputs, enabling forensic identification of model poisoning. This is supported by experiments (Table 4, rows 1-4 in the original paper).

A.4.2 Experiments

All experiments can be run end-to-end via the Python script:

```
cd FIRA_AE/AE/
python3 run_all.py
```

Alternatively, each experiment can be run individually as described below.

(E1): Firmware Causal Graph Validation [1 minutes]:
Validates that the extracted PX4 v5x firmware causal graph matches the ground truth before and after pruning.
Preparation: Ensure the conda environment is activated and navigate to `FIRA_AE/AE/v5x/BCG/`.
Execution: `cd FIRA_AE/AE/v5x/BCG`
`python before_check.py`
`python prune_check.py`
Results: `before_check.py` reports the full graph SYS, GT, ED between the extracted firmware causal graph and ground truth. `prune_check.py` reports the SYS, GT, ED after pruning by topic list, along with any nodes identified but not found. Both scripts print node counts and ED scores to stdout.

(E2): Neural Network Causality Analysis (MCG Extraction) [10 minutes]:
Extracts model causal graphs for the autonomous landing model using layer conductance attribution at multiple granularities.
Preparation: Ensure GPU is available and navigate to `FIRA_AE/v5x/MCG/`.
Execution: `cd FIRA_AE/v5x/MCG`
`python auto_land.py`
Results: The script produces three JSON files (`140.json`, `280.json`, `420.json`) containing the

important neuron lists per category at different top-n configurations. For each configuration, the script reports execution time, weighted matching accuracy on the test set, and neuron sharing statistics across categories.

(E3): Model Reconstruction from Mission [2 minutes]: Reconstructs updated models by applying mission-collected neuron data to the baseline model.
Preparation: Navigate to `FIRA_AE/`.
Execution:
```
cd FIRA_AE/AE
python model_update.py
```
Results: Three updated model files are saved to `FIRA_AE/AE/updated_model/`: `updated_50.pth`, `updated_100.pth`, and `updated_150.pth`, corresponding to models updated with 50, 100, and 150 mission-collected shared neuron groups.

(E4): Poisoned vs. Clean Model Evaluation [5 minutes]: Evaluates the baseline clean model and the poisoned model on both clean and poisoned test datasets to establish baseline accuracy differences.
Preparation: Navigate to `FIRA_AE/AE/`.
Execution:
```
cd FIRA_AE/AE
python clean_model_evaluate.py
python poison_model_evaluate.py
```
Results: Each script prints clean and poisoned accuracy for the model and saves results to CSV files in `FIRA_AE/AE/evaluation_results/`. The clean model should show similar accuracy on both datasets, while the poisoned model should show divergent accuracy.

(E5): Updated Model Evaluation [10 minutes]: Evaluates the reconstructed (updated) models on clean and poisoned datasets to assess the effect of online learning with mission data.
Preparation: Ensure (E3) has been completed so that updated models exist in `FIRA_AE/AE/updated_model/`.
Execution:
```
cd FIRA_AE/AE
python updated_model_evaluate.py
python updated_clean_model_evaluate.py
```
Results: `updated_model_evaluate.py` evaluates the three updated models (from poisoned mission data) on both datasets. `updated_clean_model_evaluate.py` updates the clean model with clean neuron data and evaluates the result. Results are printed to stdout and saved to CSV. Comparing these with (E4) demonstrates how mission-collected neuron updates affect model behavior and how forensic analysis can distinguish poisoned from clean updates.

(E6): Model Attribution Analysis (Optional) [30-50 minutes]: Performs white-box pixel-level attribution analysis on the updated models to identify which input regions drive model decisions. This part is not the key contribution of the FIRA, but we provide the script to demonstrate the updated model compatible with existing white-box analysis tools.
Preparation: Ensure (E3) has been completed. Place analysis images in `AE/analysis_data/`.
Execution: When running via `run_all.py`, type `Y` at the prompt. Or run directly:
```
cd FIRA_AE/AE
python model_analysis.py
```
Results: The script produces `.npy` files in `FIRA_AE/AE/` for each updated model, containing pixel attribution 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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: ARTO: Efficient Execution Integrity Attestation for Real-Time Operation of Cyber-Physical Systems

Ruizhe Zhao[1], Cong Sun[1*], Zongzhen Li[1], Tiantian Wang[2], and Yunbo Wang[1]
[1]State Key Laboratory of Integrated Services Networks, School of Cyber Engineering, Xidian University
[2]Khoury College of Computer Sciences, Northeastern University, San Jose, CA, USA
[1]{ruizhezhao@stu.xidian.edu.cn, suncong@xidian.edu.cn}, [2]wang.tianti@northeastern.edu

A Artifact Appendix

A.1 Abstract

We present ARTO, a control-flow and data-flow protection approach that integrates control-flow attestation to enforce execution integrity for the real-time operation of cyber-physical systems, especially autopilot systems. In this artifact, we demonstrate ARTO's functionality by presenting its complete pipeline, including both offline and online phases, for the critical operation `update_xy_controller` of the ArduCopter autopilot. In the offline phase, we present 1) the prover construction with ARTO's compiler toolchain, 2) the offline measurement generation with ARTO's simulated executor, and 3) the prover-side database embedding. In the online phase, we present 4) the prover-side local control-flow verification and equivalence-class-based data-flow protection at runtime, 5) the prover-side runtime evidence collection for the protected operations, and 6) the verifier's control-flow integrity verification based on the offline control-flow evidence.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact is designed to be conducted on an isolated virtual machine provided by the authors, which will not pose a risk to any user. To avoid the risk of misuse of the control-flow hijacking and data-only attacks described in this work, we decided not to include the attacks in the artifact. This artifact evaluation will be conducted on normal executions and on debugger-based attack injections of the protected autopilot system. Moreover, ARTO's public GitHub repository does not contain any attack that could lead to similar misuse. Besides, this artifact does not collect user data or process personal information; therefore, there is no privacy leak or misuse of user data.

*Corresponding author.

A.2.2 How to access

The source code of ARTO is available at https://github.com/zz-fz-john/ARTO-Project. For convenience, we recommend that this artifact evaluation use the pre-configured virtual machine image released at https://doi.org/10.5281/zenodo.20355901.

A.2.3 Hardware dependencies

The experiments of this artifact do not require any specialized hardware. Our artifact has been tested on a Linux workstation equipped with the Intel Xeon CPU and 64 GB of RAM. Therefore, we recommend at least 64 GB of RAM and 90 GB of storage for the artifact evaluation. In a typical application scenario, an ARTO-protected ArduCopter should run on a Raspberry Pi 3. However, for the purpose of demonstrating only ARTO's functionality, in this artifact evaluation, we use QEMU in our VM as the autopilot's execution environment. More instructions for running the ARTO-protected ArduCopter on a Raspberry Pi 3 are provided in the GitHub repository.

A.2.4 Software dependencies

Our virtual machine for artifact evaluation requires Oracle VirtualBox (v7.2.6), which has been tested with 32 GB of guest memory allocated. In the VM image, the environment for ARTO, as well as ARTO's key components, has been properly deployed, including

- Ubuntu 20.04 LTS guest operating system,
- QEMU 8.2.10 for emulating ArduCopter's execution,
- The compiler toolchain customized on LLVM 16.0.0,
- The simulated executor for offline measurement generation,
- The rewriter for embedding the hash database into the prover binary,
- The remote verifier for control-flow attestation.

We use `gcc-linaro-6.2.1-2016.11-x86_64_arm-linux-gnueabihf` for ARM cross-compilation. The VM also provides the source code of the target autopilot (ArduPilot 4.3.0).

A.2.5 Benchmarks

None.

A.3 Set-up

Importing and launching the ARTO VM. Then, the starting point for this artifact evaluation is to use ARTO's key components to build the ARTO-protected ArduCopter (prover program) and to enforce runtime control-flow and data-flow protection, as well as control-flow attestation. In this case, the reviewers can skip all the steps in Section A.3.1.

Users who intend to deploy ARTO from the very beginning can start with a clean Ubuntu system and follow the instructions in Section A.3.1 or in ARTO's GitHub repository for the complete deployment procedure.

A.3.1 Installation

Download the repository. We recommend installing ARTO in the user's home directory to avoid configuration errors. In the following instructions, we assume the repository is located at `$HOME/ARTO`.

```
$ sudo apt install -y git curl
$ cd $HOME
$ git clone https://github.com/zz-fz-john/ARTO-Project ARTO
$ cd ARTO
```

Dependency Installation. ARTO depends on several external libraries, toolchains, and Python packages. To simplify the installation process, we provide requirement files that summarize the required dependencies. Reviewers can install these dependencies using the following commands:

```
$ cd $HOME/ARTO
$ ./build_requrement.sh
```

Build Compile Toolchain. To simplify reproduction, we provide an automated script that installs and builds ARTO's compilation toolchain. The script prepares the necessary dependencies, including `gold`, `binutils`, LLVM, and SVF. Users can run the script from the ARTO's root directory:

```
$ cd $HOME/ARTO
$ chmod +x build_llvm.sh
$ ./build_llvm.sh
$ chmod +x build_svf.sh
$ ./build_svf.sh
```

A.3.2 Basic Test

The installation of the ARTO-customized LLVM can be confirmed by invoking `llvm-config` from the LLVM build directory.

```
$ cd \$HOME/ARTO
$ ./llvm-16.0/llvm-project-16.0.0/build/bin/llvm-config \
    --version
```

A successful installation should return `16.0.0`.

A.4 Evaluation workflow

A.4.1 Major Claims

In the following claims, C1 declares ARTO's offline functionality, while C2 and C3 describe the online functionality.

(C1): Offline functionality. ARTO's compiler toolchain can automatically analyze and instrument the protected operations in real-time cyber-physical systems to construct the prover program. ARTO's simulated executor can generate the offline measurements. ARTO's rewriter can embed the hash database into the final prover binary.

(C2): Normal execution and attestation. ARTO can collect compact runtime evidence of the protected operations at runtime. ARTO's remote verifier can verify the prover's execution integrity based on the control-flow offline measurement collected by the simulated executor.

(C3): Attack detection and diagnosis. ARTO can perform local control-flow verification, equivalence-class-based data-flow protection, and remote control-flow violation diagnosis at runtime.

A.4.2 Experiments

(E1): Offline functionality [15 compute-minutes + 5GB disk]:

How to: Run the following commands.

```
$ cd $HOME/ARTO
$ ./compile_0_step.txt
$ ./compile_1_step.txt
$ ./compile_2_step.txt
$ ./compile_3_step.txt
$ ./compile_4_step.txt
```

These steps insert the local verify engine, use may-alias analysis to group variables into equivalence classes, and instrument their def- and use-sites with the definition-table update or checking operation to enforce execution integrity. This procedure also identifies control-flow-related conditional branches within the critical operation.

```
$ ./compile_5_step.txt
```

`compile_5_step` invokes ARTO's LLVM backend pass to insert control-flow measurement and SFI-related code at the assembly level.

```
$ ./compile_6_step.txt
```

`compile_6_step` launches the simulated executor to generate the hash database and hash-trace map.

```
$ ./compile_7_step.txt
```

`compile_7_step` rewrites the ArduCopter binary to embed the hash database into its reserved section.

Results: The resulting binary is located at `$HOME/ARTO/ardupilot/build/SITL_arm_linux_gnueabihf/arducopter_ARTO`. To confirm that ARTO has inserted the local verify engine and data-flow protection functionality, reviewers can inspect the assembly code of ArduCopter:

```
$ arm-linux-gnueabihf-objdump -s -d \
```

```
arducopter_ARTO > arducopter_ARTO.S
```

In the assembly, calls to `checkpoint` serve as local control-flow integrity verifications. The calls to `def_check` and `use_check` implement the runtime data-flow protection for security-critical definitions and uses. The presence of these calls confirms that ARTO's control-flow and data-flow protection mechanisms have been successfully inserted into the target ArduCopter binary.

(E2): Normal execution, evidence collection, and remote verification [10 compute-minutes]:

How to: Execute a takeoff mission of the ARTO-protected ArduCopter in QEMU. During execution, the instrumented code automatically records the runtime control-flow events of the critical operation `update_xy_controller`. Conduct the following steps:

1. On the host VM, enter the QEMU environment, configure the prover, and start QEMU:
   ```
   $ cd ~/qemu-emlid-system
   $ ./config.txt
   $ ./start.sh
   ```
   Log in to the guest system in QEMU with the username `pi` and the password `raspberry`.
2. Run the instrumented ArduCopter and execute the takeoff mission. In the QEMU guest system, launch ArduCopter:
   ```
   $ ./arducopter_ARTO -S --model + --speedup 1 \
       --defaults ./copter.parm -I0
   ```
   Open another terminal in the host VM and start MAVProxy:
   ```
   $ mavproxy.py --master tcp:127.0.0.1:5760 \
       --out 127.0.0.1:14550 --console --map
   ```
   Wait until the message `AP: EKF3 IMU0 is using GPS` appears in the MAVProxy console. Then, execute the following commands in the MAVProxy console:
   ```
   STABILIZE> mode guided
   GUIDED> arm throttle
   GUIDED> takeoff 15
   ```
   Allow ArduCopter to run for approximately 30 seconds after takeoff. Then, press `Ctrl+C` in the QEMU terminal to terminate ArduCopter and complete the online evidence-collection process.
3. Run the remote verifier in the host VM.
   ```
   $ cd $HOME/ARTO/ardupilot
   $ scp -P 10022 \
       pi@127.0.0.1:/home/pi/branch_trace.txt \
       $HOME/ARTO/ardupilot
   $ scp -P 10022 \
       pi@127.0.0.1:/home/pi/hash_value.txt \
       $HOME/ARTO/ardupilot
   $ ./remote_verify.sh
   ```

Results: Upon the completion of the remote verifier's verification, the host VM terminal displays `All segments verification succeeded!`.

(E3): Local attack detection against simulated data manipulation attack injected by debugger [5 compute-minutes]:

How to: We use a GDB script to modify the runtime state of ARTO-protected ArduCopter, thereby simulating a data-manipulation attack. Specifically, we modify the runtime value of the critical variable `curr_pos` immediately before the use-site checking function is invoked. Run the following commands to launch ArduCopter in QEMU:

```
$ gdb --args ./arducopter_ARTO  -S --model + \
    --speedup 1 --defaults ./copter.parm -I0
(gdb) source modify_r5.gdb
(gdb) r
```

Then, follow the commands of **(E2)** in the host VM to start MAVProxy and control the drone to take off.

Results: The execution of ArduCopter will then be terminated by ARTO's data-flow protection. Reviewers will observe ArduCopter's termination in the QEMU terminal, together with the error message `use_check data flow error!`.

(E4): Local attack detection and remote diagnosis against simulated control-flow hijacking attack injected by debugger [5 compute-minutes]:

How to: We use a GDB script to hijack ArduCopter's runtime control flow. Specifically, in `Update_xy_controller`, we modify the return target of the last call to function `xy()` and redirect the return to another code location. First, run the commands in QEMU:

```
$ gdb --args ./arducopter_ARTO  -S --model + \
    --speedup 1 --defaults ./copter.parm -I0
(gdb) source redirect_return.gdb
(gdb) r
```

Then, follow the **(E2)** commands in the host VM to start MAVProxy and control the drone to take off. Reviewers will observe ArduCopter's termination in the QEMU terminal, demonstrating ARTO's local CFI enforcement can effectively identify the attack and stop the vulnerable execution. Furthermore, we run the remote verifier at the host VM to validate the collected execution evidence.

```
$ cd $HOME/ARTO/ardupilot
$ scp -P \
    10022 pi@127.0.0.1:/home/pi/branch_trace.txt \
    $HOME/ARTO/ardupilot
$ scp -P 10022 pi@127.0.0.1:/home/pi/hash_value.txt \
    $HOME/ARTO/ardupilot
$ scp -P 10022 \
    pi@127.0.0.1:/home/pi/target_buffer.txt \
    $HOME/ARTO/ardupilot
$ ./remote_verify_attack.sh
```

Results: The remote verifier reconstructs the execution path from the collected evidence and identifies the point at which the control-flow hijacking occurred.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix
BLE Theft Auto: Evaluating the Security of Aftermarket BLE-based Automotive Remote Control Systems

Jerry Yu* Yibo Wei* Sumanth Rao Mohak Vaswani Jefferson Chien
Christian Dameff† Nishant Bhaskar Aaron Schulman
UC San Diego

A Artifact Appendix

A.1 Abstract

This artifact package accompanies the paper *"BLE Theft Auto: Evaluating the Security of Aftermarket BLE-based Automotive Remote Control Systems"*. It includes marimo notebooks, data analysis scripts, a formal proof, a scrubbed Python KARR attack implementation, a pre-built Android BLE scanner APK, and a WiGLE data collection script. The artifacts are designed to reproduce our analyses where data sharing permits, with synthetic data used when real-world datasets cannot be released.

A.2 Artifact Availability

Two artifacts are provided: a public package (under embargo until August 12, 2026) and a restricted package containing sensitive data and analysis materials that will remain access-controlled.

A.3 Description & Requirements

The public artifact includes the notebooks `ad_cluster.py` and `scale_estimation.py`, the `wigledl.py` WiGLE collection script, the `proof.lean` formal proof, the pre-built `cluetooth.apk` Android app, and the scrubbed `karr-attack/karr/` Python implementation with secret keys removed. The restricted artifact contains sensitive data and analysis materials that cannot be distributed publicly. The notebooks are intended for artifact evaluation. `proof.lean` formalizes the algebraic core of the scale-estimation argument: under independent Bernoulli sampling with per-address inclusion probability p, the probability of an observed interval of size k is $(1-p)^{k-1}p$, so $P(X=1)=p$. It does not formalize the full empirical estimator or WiGLE dataset.

The restricted artifact contains two main components. The `data/` folder includes raw advertisement data from our wardriving collection and is required to run `ad_cluster.py` in the public artifact; copy the entire `data/` folder into the same directory as the notebook. The `data-analysis/` folder contains the original scripts used to produce figures and tables in the paper; these are not intended to be reproducible because of WiGLE EULA restrictions, but can serve as a reference for the analysis process. Reproducible versions of our data analysis algorithms are provided in the public artifact as marimo notebooks.

*These authors contributed equally to this work.
†Primarily affiliated with UC San Diego Health.

A.3.1 Security, privacy, and ethical concerns

Evaluators must not attempt unauthorized access or real-world attacks. The `karr-attack/karr/` Python code in the public artifact is provided for inspection only; secret keys are removed, and we ask reviewers not to attempt to reproduce the attack or use the artifact against any vehicle or deployed system. The `ad_cluster.py` notebook requires sensitive data: raw BLE advertisement packets collected with our Cluetooth scanner app. We removed all location, timestamp, and MAC address data before inclusion; please keep this data private and do not redistribute it. The `scale_estimation.py` notebook uses synthetic data because the original wardriving dataset is restricted by WiGLE's EULA. The Android BLE scanner app uploads scanning data to our database. **Warning: if you run the app with an Internet connection, we will learn your precise location and see identifiers from nearby devices (e.g., your phone or laptop Bluetooth name, or smart TV model).** Install and run it only if you are willing to share such data. Sensitive data and analysis materials are shared only via the restricted Zenodo record, while the public artifact is also access-controlled until August 12, 2026.

A.3.2 How to access

Download the public and restricted artifact packages from Zenodo:

- **Public artifact (embargoed until August 12, 2026):** https://doi.org/10.5281/zenodo.21405664. This package contains the notebooks, pre-built Android

APK, WiGLE script, formal proof, and scrubbed Python KARR attack implementation intended for public release when the embargo lifts on August 12, 2026.
- **Restricted artifact (permanently restricted):** https://doi.org/10.5281/zenodo.20656858. This package contains sensitive data and analysis materials that will remain restricted.

A.3.3 Hardware dependencies

A 64-bit machine is required; 8+ GB of RAM is recommended for running the notebooks smoothly. The BLE scanner app requires an Android device running Android 7.0 (API 24) or newer.

A.3.4 Software dependencies

None beyond the items listed in the Set-up section below. We recommend a Linux environment for running the artifacts.

A.3.5 Benchmarks

None.

A.4 Set-up

For a more readable walkthrough, you can also consult the `public/README.md` file included in the package.

After access is granted, download the public and restricted artifact packages and unzip both archives into the same parent directory. The archives contain top-level `public/` and `restricted/` directories. Navigate to the `public` directory; all subsequent setup and execution steps assume the current working directory is `public/` unless otherwise noted.

We use `uv` (https://docs.astral.sh/uv/) for dependency and Python version management. Install it with:

```
curl -LsSf https://astral.sh/uv/install.sh
| sh
```

The pre-built Android APK is included, so Android Studio is not required for artifact evaluation. The WiGLE data collection script requires a WiGLE API key from https://api.wigle.net/; set it in the `WIGLE_API_KEY` environment variable.

A.4.1 Installation

Set up the environment from the `public` directory:

```
uv sync
```

This automatically installs the correct Python version (3.13) and all dependencies, including the marimo notebook framework. No manual virtual environment setup is needed.

Alternatively, the public artifact includes a `Dockerfile` and `docker-compose.yml`. After copying the restricted `data/` directory as described below, start the containerized notebook environment with:

```
docker compose up -build
```

Then open http://localhost:2718 and select either notebook. The scale estimation notebook can run without the restricted data directory.

Run notebooks: Open a notebook in your browser with:

```
uv run marimo edit ad_cluster.py
uv run marimo edit scale_estimation.py
```

Android app: Install the pre-built `cluetooth.apk` on your device. See the privacy notice in the Security, privacy, and ethical concerns section before running the app.

WiGLE script (optional): If you have a WiGLE account, set `WIGLE_API_KEY` in your environment to download real data. This is not required for evaluation because the notebooks already demonstrate the algorithms using synthetic data that mimics real-world MAC address allocation patterns. If you do not have a key, skip this step.

```
export WIGLE_API_KEY="your_api_key"
uv run python wigledl.py karr.jsonl
-country=US -namelike="QT %"
```

We provide this utility to collect Bluetooth Search data from WiGLE and write raw API responses as JSONL; this is a convenience for reviewers to obtain WiGLE data in the format used by our analysis, since we cannot distribute the original dataset.

Restricted data for clustering: To run `ad_cluster.py`, copy the `data` directory from the restricted artifact into the current directory:

```
cp -r ../restricted/data .
```

This makes `data/raw_ads.txt` available alongside the notebook. If the data is missing, the notebook will display an error message with instructions.

A.4.2 Basic Test

From the `public` directory, run:

```
uv run marimo edit scale_estimation.py
```

Execute all cells. Successful execution produces the expected figures and tables without errors. For the Android app, install and run `cluetooth.apk` on an Android device and verify that BLE advertisements are detected and logged by the app.

A.5 Evaluation workflow

A.5.1 Major Claims

(C1): The advertisement clustering notebook reproduces the cluster counts and BLE name-pattern summaries re-

ported in Table 1 and supports the discovery methodology described in Section 3.1 of the paper. This claim is evaluated by Experiment E1.

(C2): The scale-estimation notebook exercises the methodology in Section 6, including the sequential address-allocation behavior illustrated in Figure 6 and the estimator used to derive the deployment estimates in Table 5. The original WiGLE-derived dataset cannot be redistributed, so Experiment E2 validates the estimator workflow with synthetic data rather than reproducing the exact values in Table 5.

(C3): The Android BLE scanner app implements the BLE advertisement collection workflow described in Section 3.1 of the paper. Experiment E3 validates that the app detects and logs nearby BLE advertisements.

A.6 Experiments

(E1): *Advertisement Clustering [30 human-minutes + standard PC]:* Reproduce the clustering results for Section 3.1 using `ad_cluster.py`.
Preparation: Run `uv sync` as described in Set-up. Copy the restricted data:

```
cp -r ../restricted/data .
```

Execution: Run:

```
uv run marimo edit ad_cluster.py
```

Execute all cells in the notebook.
Results: Compare the notebook's terminal-cluster summary with Table 1 of the paper. Confirm that the cluster counts and BLE name-pattern summaries correspond to the rows reported in that table.

(E2): *Scale Estimation with Synthetic Data [30 human-minutes + standard PC]:* Validate the scale estimation workflow for Section 6 using `scale_estimation.py`.
Preparation: Run `uv sync` as described in Set-up.
Execution: Run:

```
uv run marimo edit
scale_estimation.py
```

Execute all cells. The notebook demonstrates the full workflow using synthetic data. The last few cells optionally load real WiGLE data; if you do not have a WiGLE API key, simply skip them. The synthetic data cells are sufficient to evaluate the workflow, but not to reproduce the exact WiGLE-derived scale estimates in the paper.
Results: Verify that the notebook outputs the expected synthetic scale estimates and sequential-allocation plots consistent with Figure 6 and the Section 6 methodology. These results validate the estimator used for Table 5 but do not reproduce Table 5's exact WiGLE-derived values.

(E3): *BLE Data Collection with Android App [30 human-minutes + Android device]:* Validate the BLE scanning workflow used in the paper.
Preparation: Install `cluetooth.apk` on an Android device.
Execution: Launch the app and perform BLE scanning in your environment. **Warning: running the app with Internet access will share your precise location and nearby device identifiers with us.** If you do not wish to share nearby BLE data, stop after installing the app and do not run it, per the privacy notice in the Security, privacy, and ethical concerns section.
Results: Verify that BLE advertisements are detected and logged by the app as expected.

A.7 Notes on Reusability

The notebooks use marimo, a reactive Python notebook framework that stores notebooks as plain `.py` files. This makes them easy to version-control, diff, and integrate into existing workflows. The notebooks are designed with functional programming in mind: core routines are pure, have no side effects, and avoid context dependencies to maximize reusability on new datasets. In `scale_estimation.py`, the estimator function takes a sequence of serial numbers (integers) and returns an estimated count; any incrementing identifiers representable as integers can be adapted to this estimator, not just MAC addresses. In `ad_cluster.py`, the pipeline accepts raw advertisement packets encoded as hex strings (one per line) and runs the clustering algorithm. The hierarchical clustering function is pure and composable: future researchers can provide any distance function that takes two advertisement packets and returns a float distance, then compose new clustering pipelines from those building blocks.

A.8 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Hop: A Modern Transport and Remote Access Protocol

Paul Flammarion
Stanford University

George Hosono*
Georgia Tech

Wilson Nguyen
Stanford University

Laura Bauman
Stanford University

Daniel Rebelsky
Stanford University

Gerry Wan
Stanford University

David Adrian
Independent

Zakir Durumeric
Stanford University

A Artifact Appendix

A.1 Abstract

We provide artifacts supporting our claims on the functionality and performance of our Hop reference implementation. Our artifact includes the full client and server source code, scripts for controlled simulation experiments, and measurement scripts used to evaluate session establishment latency, file transfer throughput, and keystroke latency. The artifact also includes the datasets and plotting scripts to reproduce all performance figures presented in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Our software requires the user to install external dependencies. Our remote communication protocol can be used in a local network as well as on the public Internet. The use of hidden mode should be preferred as a means of connection for Internet-exposed Hop servers. Do not use Hop to grant access to anyone you do not trust. The code does not execute any destructive steps, and there are no ethical concerns or risks for the evaluators.

A.2.2 How to access

The artifact is hosted at Zenodo and can be accessed at: https://doi.org/10.5281/zenodo.17953396

A.2.3 Hardware dependencies

To ensure correct building and execution of our experiments, we recommend using at least two machines (physical or virtual), each with a minimum of 2 vCPUs, 4 GB of RAM, and 8 GB of storage.

*Work conducted primarily at Stanford University.

A.2.4 Software dependencies

To run all the experiments, you will need a Linux-based machine. We suggest using Ubuntu 24.04 LTS with the following dependencies:

- `Golang v1.23`, for running and building Hop. Installation available at https://go.dev. Example installation: `apt install golang`
- `Docker v28`, for building test environments. Installation available at https://docs.docker.com
- `Python3`, for evaluation only. Installation available at https://www.python.org. Example installation: `apt install python3`
- `OpenSSH v9.7p1`, for evaluation baseline. Installation available at https://www.openssh.org. Example installation: `apt install openssh-client openssh-server`
- `Rsync v3.3`, for file transfer. Installation available at https://github.com/RsyncProject/rsync. Example installation: `apt install rsync`
- `GNU Make v3.81`, for script automation. Installation available at https://www.gnu.org/software/make. Example installation: `apt install make`
- `Mininet v2.3.0`, for the evaluation in simulation. Installation available at https://mininet.org. Example installation: `apt install mininet`
- `Java 11`, to run Typometer software only used for keystroke timing evaluation. Installation available at https://www.oracle.com/java/technologies/javase/jdk11-archive-downloads.html. Example installation: `apt install openjdk-11-jre`

To ensure proper execution of Hop, we recommend using `Go 1.23` for the project, as an issue with Hop's use of `creack/pty` may prevent a shell from opening when running on newer versions of Go.

A.2.5 Benchmarks

Our artifact includes the dataset of our SSH evaluation, serving as a benchmark for Hop evaluation under the same environment. However, since the evaluator setup will likely be different (e.g., different machines, different Internet infrastructure, different times, etc.), the artifact also provides a guide to collect this data from the evaluator's setup.

A.3 Set-up

Please be sure that the machines used for evaluation have installed all software dependencies and granted permission to listen on the port or socket specified in Hop configuration (e.g., port `77` or `7777`).

When generating credentials, make sure to specify the IP address of the remote machine and to report it correctly in the client configuration file. If DNS resolution is not available for these IP addresses, use the IP address itself as both the DNS name and the server name.

Since our experiments use SSH measurements as a baseline, you may set up a working SSH communication channel between the clients and a remote SSH server for the time to shell (E1) and the file transfer in real-world (E3), and a local SSH server for file transfer in simulation (E2) and the keystroke latency (E4) experiments.

A.3.1 Installation

1. Download the Hop artifact from https://doi.org/10.5281/zenodo.17953396 and unarchive the file if required.
2. Go to Hop's directory `cd hop-go/`
3. Fetch dependencies `go get ./...; go get -t ./...`
4. Create the default credentials to locally connect to the Docker container configuration `make cred-gen`
5. Build and start the Docker container `make serve-dev`

A.3.2 Basic Test

Be sure you have a Hop server up-and-running in the previously created Docker container.

1. Connect to the Docker container `go run hop.computer/hop/cmd/hop -C containers/client_config.toml user@127.0.0.1:7777`
2. Success is having a shell as `user` in the Docker container `example.com`.

While this strictly describes a basic Hop shell access, more configurations can be found in the artifact `README` files.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *Time to shell:* Hop starts a non-interactive shell and exits with fewer network round-trips than SSH due to its handshake efficiency. This is proven by the experiment (E1) described in Section 8.2, whose results are reported in Figure 6.

(C2): *File Transfer Speed:* Hop claims having a functional congestion control in both simulation and real-world experiments. This is proven by the experiment (E2) and (E3) described in Section 8.2, whose results are reported in Figure 7 and Figure 8.

(C3): *Keystroke latency:* Hop shows a faster keystroke response time through an interactive shell than SSH. This is proven with the experiment (E4), which is described in Section 8.2 and shown in Figure 9.

A.4.2 Experiments

Each experiment requires a running Hop instance, and setting up a fully functional environment takes approximately 15 human-minutes per machine. As this is a one-time setup shared across all experiments, we do not include these initial 15 minutes in the reported time for each experiment below.

- Generate client and server credentials as specified in `hop-go/CONFIGURATION.md` or reuse the ones created for the Docker basic test. For (E2), create experiment-specific credentials with `hop-go/artifact/simulation/config/cred-gen-simulation.sh` script or manually.
- Start a Hop Hidden Docker container with `make serve-dev-hidden` or configure it manually to evaluate Hop Hidden mode.
- Generate file transfer files with random data by running `dd if=/dev/urandom of=100MB_file bs=1M count=100`. Reproduce for `10MB` and `1GB` files.

Most of the paths within the measurement scripts are relative to the root of the repository. You can adapt the paths to your environment. Each experiment's dataset can be found in the `hop-go/artifact` folder.

(E1): *[Time to shell] [15 human-minutes + 20 computer-minutes]:* Measures the time to perform a full handshake and a session setup with a remote machine for each protocol.
How to: Run the Python script to automatically measure connections and collect the session connection data.
Preparation: Have at least one server to which you can connect with the desired protocol (Hop Discoverable, Hop Hidden, and/or SSH). Update the config-

uration at the top of the `tts_measure.py` file in the `artifact/time-to-shell` folder.

Execution: Run the `tts_measure.py` script, which will output a `CSV` file. By default, the script runs 10 experiments. In the paper, we ran 100 measures for each of the 3 protocols on each machine. This value can also be updated at the top of the file.

Results: Visualize the results with any desired software or configure and run the `tts_plot.py` script.

(E2): *[File transfer in simulation] [15 human-minutes + 15 computer-minutes]:* This evaluates file transfer performance under controlled network conditions using Mininet. It isolates the effects of bandwidth, latency, jitter, and loss on end-to-end transfer time for Hop and SSH.

How to: Run the provided Mininet-based experiment script, which automatically sets up the network topology, performs file transfers, and records results.

Preparation: Ensure Mininet is installed and can be executed with elevated privileges, as well as Golang or Hops builds. This might require editing the sudoers file. Be sure that you have all the credentials for both Hop and SSH correctly located according to their respective configurations, including the `.hop/authorized_keys` file on Hop's server side. Experiment parameters such as queue size, bandwidth, delay, and file sizes can be adjusted at the top of the simulation script.

Execution: Execute the simulation script to initialize the Mininet topology, run the file transfer experiments for each configured network condition, and store the results in a `CSV` file.

Results: The output can be visualized using the `simulation_plot.py` script.

(E3): *[File transfer in real-world] [15 human-minutes + 2 computer-hour]:* Measures end-to-end file transfer time to a remote host in a real-world deployment. Transfers are performed for multiple file sizes (10 MB, 100 MB, and 1 GB).

How to: Run the measurement script, which performs repeated file transfers using `rsync` over SSH and Hop and records the elapsed time.

Preparation: Generate the files that you want to transfer during the experiment. Ensure that the target host is reachable and running the required SSH and/or Hop services. Configure host addresses, users, data files, and Hop configuration file paths at the top of `transfer_measure.py`.

Execution: Execute `transfer_measure.py` and monitor the file transfers to ensure their completion. On a separate execution, the TCP congestion control algorithm can be changed (e.g., Cubic to NewReno). You can change the congestion control of the machine to NewReno with `sysctl net.ipv4.tcp_congestion_control=reno`

Results: Results can be visualized using the `transfer_plot.py` script.

(E4): *[Keystroke latency] [30 human-minutes + 1 computer-hour]:* Evaluates interactive latency by measuring keystroke-to-echo delay over SSH and Hop sessions.

How to: Establish an interactive terminal session with the target host using either SSH or Hop, then use the Typometer third-party software to record keystroke latency for the active terminal window.

Preparation: Ensure that an interactive SSH or Hop session can be established with the target host. The Typometer software and its license are provided in the `keystroke-latency` directory. Follow the included Typometer 'README' to install and launch Typometer. Ensure that you are using Java 11 to start Typometer. Keep the default Typometer settings.

Execution: Start Typometer and select the active terminal window. Typometer will write a keystroke and automatically measure the time to its display. Repeat the measurement for each protocol and host under identical conditions. Export the recorded latency data as a `CSV` file.

Results: Results can be visualized using the `keystrokes_plot.py` script.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: <WAVED: Principled Identification of Off-Path Exploitable Weak Verifications within the TCP/IP Protocol Suite>

Yizhou Zhao[1], Xuewei Feng[1]✉, Min Li[2], and Ke Xu[1,2]✉

[1]Tsinghua University, Beijing, China
[2]Zhongguancun Laboratory, Beijing, China
{zhaoyz24@mails.,xuke@}tsinghua.edu.cn, fengxw06@126.com, limin@mail.zgclab.edu.cn

A Artifact Appendix

A.1 Abstract

This artifact provides the complete implementation of WAVED and the experimental environment required to reproduce the results presented in our paper. It specifically supports the reproduction of our main experiments, which identify weak-verification paths to risky operations across different kernel stacks, and the ablation studies that demonstrate the effectiveness of our reduction in path discovery. To ensure ease of deployment and consistent reproducibility, we provide a Dockerfile that automatically constructs the necessary build environment. The artifact also includes comprehensive step-by-step instructions, sample outputs, and detailed interpretations of the results to facilitate the evaluation process.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Since WAVED is built upon static program analysis, all experiments listed below can be conducted on an isolated machine. The entire evaluation process is non-destructive.

A.2.2 How to access

We released the complete implementation of WAVED, with the user guidance and necessary scripts to reproduce all the results at https://doi.org/10.5281/zenodo.17896119.

A.2.3 Hardware dependencies

A server equipped with an x86-64 CPU and a minimum of 350 GB of memory is required to complete all experiments without memory exhaustion. To speed up building the dependencies, we recommend using a processor with 8 or more cores. We validated our artifact on a system featuring an Intel Xeon Gold 5418Y processor and 500 GB of RAM.

A.2.4 Software dependencies

A Linux-based operating system is required to run the artifact. We specifically recommend Ubuntu 22.04 or 24.04 for optimal compatibility. The other dependencies are listed in the *README.md* and *README-exp.md* in the repository.

A.2.5 Benchmarks

The artifact is evaluated on real-world operating system kernels acting as macro-benchmarks. Specifically, the static analysis is conducted on the source code of Linux 5.15, Linux 6.8, and FreeBSD 14.1, released at https://github.com/torvalds/linux.git and https://github.com/freebsd/freebsd-src.git. By applying our analysis to these complex systems, we demonstrate the artifact's practical capability in consolidating and strengthening the code quality of real-world protocol stacks. For convenience, we provide precompiled bitcode files of the IPv4/IPv6 implementations of these kernels in *bcfiles/* directory.

A.3 Set-up

A.3.1 Installation

We provide a building script (*build.sh*) to automatically build the artifact environment. For convenience, we also provide a Dockerfile to automatically construct the necessary environment and components for WAVED. This process can be executed by running the *build-docker.sh* script. After the Docker image is built, *run-docker.sh* will create a Docker container and open the shell.

A.3.2 Basic Test

Build the Docker image and start the container by following the instructions in *README-exp.md*. Next, execute *experiments/run-basic-test.sh* inside the container. If the output displays "`All components work fine. Go ahead`

`to run experiments!"`, the basic tests have passed successfully.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): WAVED reports 19 weak-verification vulnerabilities in Linux 5.15, Linux 6.8 and FreeBSD 14.1 with considerably high precision. This is proven by the experiment (E1) described in Section 6.4, and more details are categorized in Section 7. The results are illustrated in Table 5, 6, and 7.

(C2): WAVED contributes a significant reduction (a total of 89.3% in average) comparing to traditional taint analysis in discovering weak-verification paths. This is proven by the experiment (E2) described in Section 6.4. The results are illustrated in Table 8.

A.4.2 Experiments

(E1): Main Experiment: [around 4 compute-hour]: The normal workflow of WAVED in weak-verification path discovery.
Preparation: Build the Docker image by executing the script *build-docker.sh*. Then run *run-docker.sh* to start a Docker container and access the shell.
Execution: Run *experiments/run-main-exp.sh* in the Docker container.
Results: Results can be found in *docker_results/* directory. *docker_results/main_results.txt* contains the discovered path numbers of each type. *docker_results/output-paths/* contains the details of each found path.

(E2): Ablation Experiments [around 19 compute-hour]: Ablation study on WAVED's reduction efficiency. Configurations include disabling strength filtering after constraint deduplication (w/o SF), and comparison with traditional implicit taint analysis (w/o DT).
Preparation: The same process as in E1.
Execution: Run *experiments/run-ablation-exp.sh* in the Docker container.
Results: Results can be found in *docker_results/* directory. *docker_results/ablation_results_woSF.txt* contains the discovered path numbers under the [w/o SF] configuration. *docker_results/ablation_results_woDT.txt* contains the discovered path numbers under the [w/o DT] configuration.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Download More RAM: Dismantling Windows Operating System Defences with Mischievous Memory

Sam Collins
University of Birmingham

Tom Chothia
University of Birmingham

William Burgess
University of Birmingham

Marius Muench
University of Birmingham

David Oswald
Durham University

A Artifact Appendix

A.1 Abstract

Download More RAM describes an attack methodoly to subvert Windows defenses by introducing aliased memory from software, assuming the presence of DIMMs without SPD write protection. The artifact in this document describes how to reproduce the attack, including (i) checking whether installed DIMMs are susceptible to the attack, (ii) introducing and reading out aliased memory, (iii) patching Windows code integrity to enable loading of previously block-listed vulnerable drivers, and (iv) disabling AV and EDR software.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The attack interferes with the operation and security of the Windows operating system. Thus, data on the system may be at risk; unsuccessful runs of the attack may corrupt the system. We do not recommend to try out the attack on a live system with user data; all our experiments have been carried out on a test machine with a fresh Windows installation.

A.2.2 How to access

The development repository for *Download More RAM* is at: https://github.com/SamCollins1327/Download-More-RAM.

We also provide a stable version of the artifact at: https://doi.org/10.5281/zenodo.20331553.

A.2.3 Hardware dependencies

The full *Download More RAM* attack requires DDR4/DDR5 DIMMs without SPD write protection, being installed on a Secure Boot enabled platform.

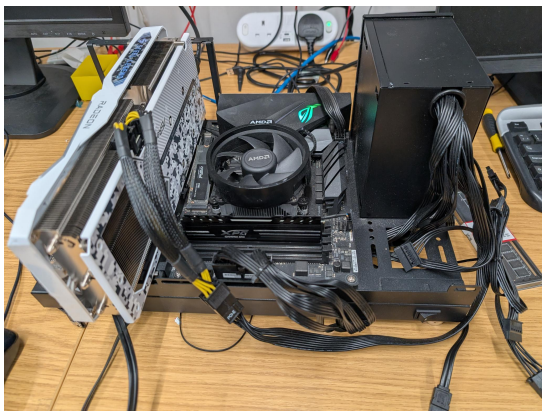


Figure 1: The experiment platform used for developing *Download More RAM*.

Known vulnerable setup. We developed *Download More RAM* on a ROG Strix B450F motherboard, with a Ryzen 7 2700 CPU, and a single 8GB stick of [Corsair Vengence, ADATA XPG, G.Skill Aegis] DDR4 memory.

Our setup is shown in Figure 1. Other hardware setups may require minor adjustments to the attack. For the artifact evaluation, we will provide assistance for getting the attack running on other hardware platforms.

A.2.4 Software dependencies

Our attack targets Windows 11, with the artifact specifically targeting build 28000.1. All tooling used is available in our artifact repository with the exception of Windows 11, and Primo Ramdisk. Windows 11 is free to use without a license and Primo Ramdisk offers a 30 day trial version.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Clone (or download) the artifact repository:

```
git clone git@github.com:SamCollins1327/Download-More-RAM.git
```

A.3.2 Build Instructions

We provide pre-compiled tools for ease of use. To build tools from source follow these steps.

- For *DMR_SPD_Tool*, *DMR_SPD_Tool_Headless*, *RTCore-write*, *RTCore_dump*, *RTCore_read*, *TrueSightKiller*, *TrueSightKiller_NoDriverLoad*, and *WinPmem-custom*: These tools are provided as visual studio projects. They can be built using Visual Studio Community Edition. For each project, open the corresponding .sln file, navigate to the build menu and build as release.
- For *helper_optimised.c*, *patcher.c*, and *sig_finder.c*: These tools can be compiled using gcc with no additional options with Mingw-w64.

A.3.3 Basic Test

On the system, run the `DMR_SPD_Tool` executable from an elevated command prompt. Select the DIMM slot corresponding to your installed DIMM and then select the option to check if SPD write protection is enabled (*"2. Check write protection"*). If the result indicates that SPD write protection is not enabled the the system is vulnerable to the Downlaod More RAM attack, assuming a pre-mitigation version of Windows 11 is running (i.e., any version before 26200.8246 and 26100.8246). If the result indicates that SPD write protection is enabled, then the DIMM SPD cannot be written from software and the system can be considered safe.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): A range of consumer DIMMs do not utilise write protection on their SPD chips, allowing to launch *Download More RAM* attacks from software. Experiment *(E1)* provides a custom testing utility for a DIMM's write protection status. We present our testing results in Table 2 of the paper.

(C2): *Download More RAM* allows introducing memory aliasing from a privileged user on Windows. Experiment *(E2)* replicates aliasing on a machine with DIMMs without SPD write protection and reads out contents of the aliased DIMM. The methodology is described in Section 6.1 and 6.2 of the paper.

(C3): Aliased memory can be re-written, ignoring privilege boundaries and memory protections. Experiment *(E3)* demonstrates the write primitive, on the example of patching the Windows code integrity module *skci.dll* in memory. This is described in Section 6.3 and 6.4 of the paper.

(C4): After disabling code integrity checks, *Download More RAM* allows to load previously blocked drivers, enabling further attacks. We demonstrate this in *(E4)* on the example of running TrueSightKiller, a tool disabling AV and EDR software. This corresponds to case study *"Disabling PPL protected AV/EDR"*, described in Section 7.2 of the paper.

A.4.2 Experiments

(E1): *[Verifying SPD Vulnerability] [30 human-minutes + negligible compute time + 1GB disk]:*
Preparation: On the target system running Windows 11, with secure boot and core isolation enabled (ideally with motherboard or equivalent chipset as described in A.2.3) install candidate DIMM. Have the precompiled `DMR_SPD_Tool` executable on disk ready.
Execution: Run the `DMR_SPD_Tool` executable from an elevated command prompt. From here select the DIMM slot corresponding to your installed DIMM, then select the option to check if SPD write protection is enabled (*"2. Check write protection"*).
Results: If the DIMM SPD is not write protected the SPD tool will print the message "NOT write protected". This means the DIMM is vulnerable to software induced aliasing. Conversely if the DIMM SPD is write protected the message will include "WRITE PROTECTED" and can be considered safe.
Note: The `DMR_SPD_Tool` was built on/for our target system and we have not verified it on other chipsets. Therefore, if the hardware is not avaliable then the Thaiphoon Burner v174 utility can be used for this step on a more universal set of systems, but not for SPD writing as is required in E2 without the paid license. SPD write protection can be checked via Thaiphoon Burner's *EEPROM->Data Protection Check* menu.

(E2): *[Alias Memory] [30 human-minutes + negligible compute time + 1GB disk]:*
Preparation: On the target system running Windows 11 (ideally with motherboard or equivalent chipset as described in A.2.3) install a DIMM verified to have no SPD write protection (e.g., the 8GB DDR4 DIMMs by Corsair Vengence, ADATA XPG, or G.Skill Aegis). Have the precompiled `DMR_SPD_Tool` executable on disk ready.
Execution: Run the `DMR_SPD_Tool` executable from an elevated command prompt. Select the DIMM slot corresponding to your installed DIMM, then select the option to alias the selected DIMM (*"3. Alias DIMM"*).

From an elevated command prompt run `bcdedit /set current removememory 8192` (assuming an 8GB DIMM) to prevent Windows from using the newly aliased memory. Reboot the system. This action is reversible via option (*"4. Un-alias DIMM"*).

Results: Upon reboot the system should report double the installed memory as is physically present i.e. 8GB->16GB. This can be verified both in the BIOS and in the msinfo utility in Windows.

(E3 & E4): *[Patching skci and killing Windows Defender] [30 human-minutes + negligible compute time + 18GB disk]:*

Preparation: On the target system, set up with aliased memory as described in E1 and E2, install Primo Ramdisk utility trial version and add the path to rxprd.exe in the installation folder to the PATH environmental variable.

Execution: Navigate into the Precompiled_Tools directory. (3) Set the path values in runme.bat to the directory of the precompiled tools file. Run runme.bat. In the case the signatures are not found (a) ensure invisible memory can be enabled in Primo Ramdisk (a restart may be required if not) and (b) verify manually the location of skci.dll in the current_decoded.raw file output using one function fingerprint from the patcher.c source code. For example if the module is found in the 30th-31st MB of the system, set the Ramdisk allocation in runme.bat line 12 to 28 (as is set by default).

Results: All functions should appear with a trailing *"...OK"* in the command output, signaling success. runme.bat will attempt to run truesightkiller.exe before and after patching skci.dll. The first attempt should give a warning that the driver cannot be loaded and deliberatley fail, the second (after the patch) should disable Windows defender by terminating its process.

A.5 Notes on Reusability

Our attack was implemented on the target machine described in Section A.2.3. Other machine setups may require modification of the attack, e.g., due to different memory scrambler implementations or BIOS-level defenses against the attack.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Secpoline: A Scalable Approach to Build Secure In-Process Syscall Interposers

Ruben Sturm
DistriNet, KU Leuven

Anton Schelfhout
DistriNet, KU Leuven

Merve Gülmez
Ericsson Security Research

Adriaan Jacobs
DistriNet, KU Leuven

Stijn Volckaert
DistriNet, KU Leuven

A Artifact Appendix

A.1 Abstract

Secpoline is a high-performance secure in-process syscall interposer. The artifact concists of the source code for Secpoline and a collection of case studies and benchmarks to support our claims. The evaluation consists of four main components:

Security Evaluation: We implement proof-of-concept attacks targeting Secpoline based on a set of known attacks collected from prior work (Section 7.1).

Performance Evaluation: We evaluate the performance overhead of Secpoline compared to the state-of-the-art secure in-process syscall interposer (Section 7.2) and MPK sandbox (Section 7.3). To do so, we adapt the benchmarks used by these existing prototypes.

Falco Case Study: We implement a kernel-module-free version of the Falco intrusion detection engine and evaluate it (Section 8.1).

ProxySQL Case Study: We integrate ProxySQL into Secpoline to implement an in-process sidecar (Section 8.2).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

In general, running the code in this artifact does not pose any security or privacy risks.

The only potential concern arises when running the zip benchmark for the Falco case study, as it installs a kernel module that intercepts system calls system-wide. This is a public and well-tested kernel module provided by Falco (https://falco.org/docs/concepts/event-sources/kernel/), and it is automatically uninstalled at the end of the benchmark.

A.2.2 How to access

The source code required for this artifact is available at https://github.com/lazypoline/secpoline. This repository contains three branches, the main branch contains the source code for the baseline version of Secpoline and a branch for each case study evaluated (ProxySQL and Falco).

A.2.3 Hardware dependencies

The artifact requires an x86_64 CPU that supports Intel MPK.

A.2.4 Software dependencies

This artifact requires a Linux system with kernel version 6.12+ and CMake version 3.16+. The artifact has been tested on Ubuntu 20.04 and Ubuntu 24.04. Part of the artifact requires a modified version of the Linux kernel. This artifact contains a patch for Linux kernel version 6.12.0 to create this modified kernel.

A.2.5 Benchmarks

This artifact requires the Cerberus benchmarks (https://github.com/ku-leuven-msec/The-Cerberus-Project), the endokernel benchmarks (https://github.com/endokernel/test) and the apache httpd server (https://github.com/apache/httpd).

A.3 Set-up

Secpoline requires access to `mmap` at low virtual addresses and the ability to place hardware breakpoints. These features can be enabled using:

```
echo 0 | sudo tee /proc/sys/vm/mmap_min_addr
echo 2 | sudo tee
    /proc/sys/kernel/perf_event_paranoid
```

A.3.1 Installation

More detailed instructions can be found in the README of each branch.

To test the base functionality of Secpoline, use the code in the main branch. Start by installing the required dependencies:

```
apt install libaudit-dev musl-tools clang
    cmake libssl-dev
```

Then build Secpoline using CMake:

```
# From the root of the repository
mkdir build
cd build
cmake ../
make -j
```

Now Secpoline can be run as follows:

```
path/to/output/libloader.so
    path/to/output/secpoline <some binary>
```

A.3.2 Basic Test

A basic test example is provided that is built alongside Secpoline. To verify whether Secpoline is actually running and intercepting syscalls, enable the `POST_PRINT_SYSCALLS` option in `src/include/config.h`.

Now, from the `output` folder, run:

```
./libloader.so ./secpoline ./main
```

Secpoline can be tested on other applications as well, for example:

```
./libloader.so ./secpoline /usr/bin/ls
```

This should print all syscalls invoked by the application, including their arguments and return values. example output: `[tid 3079692] openat(0xffffff9c, 0x7fe399fd48b0, 0x80000, 0x0, 0x7fe39ac66010, 0x7) = 0x5`

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Secpoline achieves parity with the collective security guarantees of existing PKU-protected syscall interposers. This is proven by experiment (E1) described in Section 7.1 whose results are illustrated in Table 1.

(C2): Secpoline achieves similar overhead in syscall intensive applications compared to a state-of-the-art secure in-process syscall interposer and significantly lower overhead compared to a cross-process approach. This is proven in experiment (E2) described in Section 7.2 whose results are illustrated in Figure 3 and Figure 4.

(C3): Secpoline achieves competitive performance compared to a state-of-the-art MPK sandbox while avoiding the expensive cross-process fallback paths commonly required by process-isolated designs. This is proven in experiment (E3) described in Section 7.3 whose results are illustrated in Figure 5 and Figure 6.

(C4): Secpoline can transparently integrate large and complex existing codebases into its syscall interposition framework, enabling practical in-process implementations of real-world systems such as intrusion detection engines and proxy services. This is described in Section 8 and demonstrated by experiments (E4) and (E5), which integrate Falco and ProxySQL into Secpoline, respectively. In particular, (E4) shows that Secpoline can replace Falco's kernel-module-based syscall interception with a fully in-process interception approach, while (E5) demonstrates that Secpoline can virtualize inter-process proxy communication by embedding ProxySQL directly into the monitored process. The resulting systems maintain practical performance characteristics while supporting significantly more flexible in-process mediation and isolation. The results of these experiments are illustrated in Figure 7 and Figure 8.

A.4.2 Experiments

(E1): [Security Evaluation] [45 human-minutes + 30 compute-minutes]: Verification that Secpoline protects against a set of known attacks. This evaluation contains multiple proof-of-concept attacks. These should all crash with a `SIGSEGV`, `SIGILL`, or return an error code to demonstrate that the attack failed.
Preparation: Build Secpoline using CMake. This will build all proof-of-concept attacks, which can be found in `output/security_eval`. The README contains an explanation of how to set up the `Forged signal delivery` attack, as this requires some manual work.
Execution: Run each of the proof-of-concept attacks.
Results: All of these attacks should fail or terminate before completing. An attack succeeds if it outputs `"sud disabled"`.

(E2): [Nexpoline Benchmarks] [1 human-hour + 1 compute-hour]: Comparison of the overhead introduced by Secpoline, Nexpoline/Endokernel, and strace across several benchmarks, showing similar overhead between Secpoline and Nexpoline/Endokernel and significantly higher overhead for strace.
Preparation: Follow the instructions to set up the Endokernel Docker environment and update its test cases. These instructions can be found in `benchmarks/benchmark_endokernel.md`.
Execution: Run `zip.py`, `sqlite.py`, `curl.py`, and `lmbench.py`.
Results: Run `calc_overhead.py` for each of these tests. This will output the results shown in Figure 3 and Figure 4.

(E3): [Cerberus Benchmarks] [1 human-hour + 2 compute-hours]: Comparison against a state-of-the-art MPK sandbox across two HTTP servers.
Preparation: Obtain the Cerberus source code (https://github.com/ku-leuven-msec/The-Cerberus-Project) and the Apache HTTP Server source code (https://github.com/apache/httpd). Then build Cerberus, its benchmarks, and the

Apache HTTP Server.
Execution: Instructions to run both lighttpd and Apache can be found in `benchmarks/benchmark_cerberus.md`. This document contains the commands required to run the benchmarks under the different configurations described in the paper.
Results: The output of `wrk` shows the requests per second and reproduces the results shown in Figure 5 and Figure 6.

(E4): [Falco Case Study] [30 human-minutes + 15 compute-minutes]: Demonstration of Secpoline's ability to integrate large codebases and replace kernel modules for syscall interception without significant performance loss.
Preparation: Switch to the `falco_usecase` branch. After building Secpoline, update the `Secpoline_plugin` path in `falco/falco.yaml` and `falco/falco_base.yaml`.
Execution: Run the `zip_bench.sh` script in the `benchmarks/falco_benchmarks/` directory. This will install a kernel module to compare against the Secpoline interception approach.
Results: This outputs the total execution time in seconds for each run and should demonstrate the relatively minor performance cost of replacing a kernel module with Secpoline.

(E5): [ProxySQL Case Study] [30 human-minutes + 30 compute-minutes]: Demonstration of Secpoline's ability to integrate large codebases and achieve significant performance improvements through cross-process communication virtualization.
Preparation: Switch to the `proxysql_usecase` branch and build Secpoline. Ensure that `mysql` is installed. Then, in the `libproxysql/benchmark` directory, run the `init_mysql_database.sh` script, which sets up the database and user required by the benchmarks. The README also contains instructions for performing this step manually.
Execution: Run the `sysbench.sh` script.
Results: This outputs the total throughput and latency for each run.

A.5 Notes on Reusability

Secpoline is a highly customizable syscall interposer that can be easily extended to support a wide range of use cases. To adapt Secpoline for a new use case, add code to the following components:

Adding new syscall filter rules. To add new filter rules, define a new handler function in `main_monitor.cpp` using the `SYSCALL_HANDLER` macro and declare it in `secpoline.h` using the `DECLARE_HANDLER` macro. These handlers use the `gprs` class to read from and write to application registers. The functions `memcpy_untrusted_to_trusted()` and `memcpy_trusted_to_untrusted()` provide safe access to untrusted memory, which is otherwise disabled by the exclusive MPK permission policy.

Syscall handlers should always return `false`.

Assigning handlers to syscalls. Once a new handler has been defined, it must be assigned to a syscall in `secpoline.cpp:initialise_syscall_handlers()`. There are three types of handlers:

- **Prehandlers:** Any number of prehandlers can be added for a syscall. These execute before the main handler and should not invoke the syscall on behalf of the application. Instead, they may modify syscall arguments before execution.
- **Main handler:** Only one main handler can be assigned to a syscall. While prehandlers and posthandlers may invoke syscalls internally, only the main handler should emulate the syscall for the application.
- **Posthandlers:** Any number of posthandlers can be added for a syscall. These execute after the main handler and should assume that the syscall return value is already stored in `rax`.

Initialization code. Add any initialization code to `secpoline.cpp:main()` after the call to `init_page_manager()`, as this is the point where Secpoline is fully initialized and ready to support arbitrary code.

Meta-monitor. To add new syscall filter rules for monitoring syscalls, update `meta_monitor.cpp:meta_monitor()`. Do not call any monitor-unaware code from this function, and ensure that all syscalls executed during meta-monitor handling are inlined to avoid replacement with `call *rax`.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Plaintext Recovery Against Post-Filtering Access Control

Zachary Espiritu
MongoDB Research

David Cash
University of Chicago

A Artifact Appendix

A.1 Abstract

In this paper, we show how to amplify side-channels in *fine-grained access control* database mechanisms such as *row-level security* (RLS) or *document-level security* (DLS) to expressive record reconstruction by using rich query interfaces (e.g., ranges, prefixes). We implement a family of CLI utilities called `unfilter-*` which implement these attacks:

- **`unfilter-rls` (RLS).** In Section 3 of the paper, we exploit a timing side-channel and expressive SQL queries (e.g., ranges, conjunctions) to enumerate unknown attribute values in *relational databases* like PostgreSQL. We then show how to recover full records via binary search over large domains. Our tool implements these attacks.
- **`unfilter-dls` (DLS).** In Section 4 of the paper, we exploit scoring and prefix-expansion side-channels to recover indexed terms from documents in *full-text search databases* like Elasticsearch and OpenSearch. Under certain index configurations, we can also extract n-grams in the corpus to recover approximate text.

Both artifacts are designed to be extensible and usable for not just future research projects, but also for industry practitioners who are interested in seeing how applicable these attacks may be in their system. Each artifact contains documentation on how to add new evaluation system functionality (in addition to regression tests and strong code typing and formatting requirements) as well as how to extend it to other scenarios.

It is helpful to think of this artifact as two independent artifacts. We distinguish between the **RLS** and **DLS** sections in the Artifact Appendix using highlighted colors.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact runs all attack experiments on *isolated* systems, though in theory one could adapt our scripts to attack live systems. We discuss more details about our ethical analysis in the Ethical Considerations section of our paper.

For evaluators, the **RLS attacks** require a large number of network requests between the target database host and the attacker host. This will introduce a high amount of network load, so evaluators should be careful with the network environment in which they choose to reproduce our results. They also require a reasonably high degree of parallelism (up to 16 workers) and memory (up to 64 GB) which can potentially overwhelm system resources and cause crashes.

The **DLS attacks** can either be run between two host machines (over a network) or locally in Docker containers, though they also require a high amount of memory (up to 64 GB). See Section A.2.3 for details.

A.2.2 How to access

Our artifacts are accessible for permanent access on Zenodo at `https://doi.org/10.5281/zenodo.20370280`.

A.2.3 Hardware dependencies

The **RLS** experiments exploit timing side-channels whose accuracy is sensitive to noise and hardware. Evaluators are welcome to run the **RLS** experiments on their own setup, but they may get substantially different results (due to noise) unless they use equivalent architecture to the VMs described below. On the other hand, the **DLS** experiments can be run on a single host, though the largest experiments require ≈64 GB of RAM.

While the **RLS** experiments can run on any hardware, replicating our specific accuracy results requires *dedicated-core* machines like the `c4` family of Google Compute Engine (GCE) VMs that we use in the paper. Specifically:

- *Database servers* run on a `c4-standard-16` VM with 16 vCPUs, 60 GB RAM, and a 200 GB local SSD.
- The *attacker* is located on a `c4-highmem-16` VM with 16 vCPUs, 128 GB RAM, and a 100 GB disk.
- Some experiments use a *noise* `c4-standard-96` VM with 96 vCPUs, 128 GB RAM, and a 100 GB disk.

The **DLS** experiments are not sensitive to noise and can be run on any hardware that meets the memory requirements. The primary focus for the **DLS** claims is on query counts and accuracy which should be essentially the same on all setups.

Nevertheless, for consistency (and to capture reasonable wall-clock time numbers), the paper's **DLS** results were run on the following GCE VMs:

- *Database servers* run on a `c4-standard-16` VM with 16 vCPUs, 60 GB RAM, and a 200 GB local SSD.
- The *attacker* is located on a `c4-standard-16` VM with 16 vCPUs, 60 GB RAM, and a 200 GB local SSD.

A.2.4 Software dependencies

The artifact provides scripts which automatically install all necessary dependencies. These scripts have been primarily tested on clean installs of **Debian 13** and **Ubuntu 24.04**. See Section A.3.1 for installation instructions.

A.2.5 Benchmarks

The **RLS** experiments require the use of a synthetic electronic medical record dataset which is provided in the artifact. The **DLS** experiments require the use of truncated Enron datasets which are provided in the artifact. Scripts for generating these datasets are in the artifact.

A.3 Set-up

A.3.1 Installation

Installing RLS. Start by downloading the artifact from Zenodo. Then, unzip the artifact and enter the `rls` directory:

```
cd fgac-main/rls
```

Then, install all **RLS** dependencies via:

```
bash setup.sh
```

You will need to either run the `source` command printed at the end of the `setup.sh` script or restart your terminal. Finally, authenticate the `gcloud` CLI to your Google Cloud credentials:

```
gcloud auth login
```

> While we recommend the integrated Google Cloud orchestration features in `unfilter-rls` for easier reproducibility, see Section A.6 if you want to run the **RLS** artifact on machines that do not involve Google Cloud.

Installing DLS. Start by downloading the artifact from Zenodo. Then, unzip the artifact and enter the `dls` directory:

```
cd fgac-main/dls
```

Then, install all **DLS** dependencies via:

```
bash setup.sh
```

You will need to either run the `source` command printed at the end of the `setup.sh` script or restart your terminal.

A.3.2 Basic Test

Kick-the-tires on RLS. To make sure your setup is working and that your account has the ability to provision Google Compute Engine VMs, first check that your environment has been properly setup:

```
unfilter-rls doctor
```

If all rows print "OK", then you can run a quick experiment (**E-R9**):

```
unfilter-rls claims run C-R9
# also supports 'unfilter-rls claims run 9'
```

This will take about 20–25 compute-minutes to run (most of this time is spent provisioning and tearing down Google Cloud VMs). After this runs, it should write this file:

```
rls/results/dbsize/summary.txt
```

Verify that the numbers align with the statistics reported in the **"Schema and data" paragraph in Section 3.3** of the paper. If this works, you should be ready.

Kick-the-tires on DLS. To make sure your setup is working, first check that your environment has been properly setup:

```
unfilter-dls doctor
```

If all rows print "OK", then you can start running experiments. For convenience, we provide a one-shot script which runs all of the DLS experiments in sequence. To quickly test that it works, you can run the DLS attack pipeline on only the smallest dataset ($|D| = 1$):

```
bash dls/run.sh --datasets 1
```

This will take about 5 compute-minutes to run. This should produce the following files:

```
dls/results/reviewer/opensearch_table.tex
dls/results/reviewer/elasticsearch_table.tex
```

The rows should match the first $|D| = 1$ row of **Table 6** in the paper. If this works, you should be ready.

A.4 Evaluation workflow

For readability, we organize our claims and experiments by whether they are related to the **RLS (R)** or **DLS (D)** attack.

A.4.1 Major Claims

(C-R1): Authorized and unauthorized queries are reliably longer than non-existent queries under join-based policies. This is proven by the experiment (**E-R1**) described in Section 3.1 whose results are illustrated in **Figure 2**.

(C-R2): The RLS timing side-channel is preserved under expressive queries. This is proven by the experiment (**E-R2**) described in Section 3.2 whose results are illustrated in **Figure 3**.

(C-R3): For **join**-based policies, $k = 1,2,3,4$ suffices for $> 99.9\%$ accuracy at CPU $\leq 95\%$. **Inline**-based policies are significantly noisier. When load decreases, fewer queries suffice. This is proven by the experiment (**E-R3**) described in Section 3.3.1 whose results are in **Table 1**.

(C-R4): Cross-region latency impacts accuracy but can be mitigated by increasing k. This is proven by the experiment (**E-R4**) described in the "Physical distance" paragraph in Section 3.3.1 whose results are in **Table 2**.

(C-R5): *Transparent data encryption* does not change the timing differences we exploit. This is proven by the experiment (**E-R5**) described in the "Transparent data encryption" paragraph in Section 3.3.1. There are no figures in the paper directly associated with this experiment, but the experiment should produce figures that look very similar to **Figure 2** and **Figure 3**.

(C-R6): When $|D| \gg |S|$ (e.g., `ssn`), binary search yields large speedups over linear probing. When $|D| \approx |S|$ (e.g., `zip_code`, `age`), linear probing is already practical and binary search slows the attack. Parallelism speeds up the attack, but too many threads decrease performance due to increased (self-imposed) DB load. This is proven by the experiment (**E-R6**) described in Section 3.3.2 whose results are in **Table 3**.

(C-R7): Attack performs better when starting with most selective attributes first. The low selectivity of `age` and `zip` introduces timing variance which results in loss of accuracy. This is proven by the experiment (**E-R7**) described in Section 3.3.2 whose results are in **Table 4**.

(C-R8): The *subquery + composite index* approach closes the side-channel for our scenario. Adding just a composite index *or* only changing the policy to a subquery is not sufficient to mitigate the attack. This is proven by the experiment (**E-R8**) described in Section 3.4 whose results are in **Table 5**.

(C-R9): The full `patients` and `doctors` database occupies 185 MB on disk (79 MB heap, 37 MB `ssn` index, 11 MB `zip_code` index, 7 MB `age` index, 51 MB other indexes). This is proven by the experiment (**E-R9**) whose results are in the **"Schema and data" paragraph in Section 3.3**.

(C-D1): On OpenSearch, the DLS attacks recover the indexed n-gram set exactly across all tested corpus sizes ($|D| \in \{1, 10, 100, 1000\}$). Injections and queries scale linearly with the # of trie nodes. This is proven by the experiment (**E-D1**) described in Section 4.5 whose results are in **Table 6**.

(C-D2): The attack produces identical results on Elasticsearch as on OpenSearch. This is proven by the experiment (**E-D2**) described in Section 4.5. There are no figures in the paper directly associated with this experiment, but the experiment should produce results very similar to **Table 6**.

(C-D3): The de Bruijn text traversal on the 4-grams recovers approximate document text. While text is not complete, the overall meaning is preserved. This is supported by the experiment (**E-D3**) described in the "Document reconstruction" paragraph in Section 4.5 from which sample results are in **Figure 5**.

A.4.2 Experiments

(E-R_): Run all RLS experiments *[20 human-minutes + 36 compute-hours]*

For evaluator convenience, our **RLS** artifact provides a command which runs all of the **RLS** experiments in a single pipeline.

Execution: This runs the full experiment pipeline:

```
unfilter-rls claims run 1,2,3,4,5,6,7,8,9
```

The script displays frequent progress updates in the terminal. We recommend the use of a terminal detacher like `screen` if you wish to close your terminal while the script is running. (See the top-level `README.md` in the artifact for a guide on using `screen`.)

This script runs the following experiments (the per-experiment time estimates below exclude VM provisioning and teardown time, as those provisioning steps only happen once when running all experiments in the same `unfilter-rls` command):

(E-R1): Equality microbenchmark (Figure 2) *[5 compute-minutes]*

This experiment verifies that the timing side-channel is preserved with *equality queries* under the *join*-based policy described in Section 3.3.

Results: Results are written as a rendered TEX figure. The TEX source code is:

```
rls/results/existence/existence_kde.pgf
rls/results/existence/
existence_kde_figure.tex
```

Compare this to **Figure 2** in the paper.

(E-R2): Range query microbenchmark (Figure 3) *[5 compute-minutes]*

This experiment verifies that the timing side-channel is preserved with *range queries* under the *join*-based policy described in Section 3.3.

Results: Results are written to:

```
rls/results/range/existence_range_kde.pgf
rls/results/range/
existence_range_kde_figure.tex
```

Compare this to **Figure 3** in the paper.

(E-R3): CPU load vs accuracy (Table 1) *[4 compute-hours]*

This experiment checks how the accuracy of the timing oracle changes based on CPU load introduced by a 3rd *noise* VM.
Results: Results are written to:

```
rls/results/table1/table1.pdf
```

Compare this to **Table 1** in the paper—please be aware that due to rate limiting that we cannot control in this *specific* GCE environment, less load than anticipated may occur on higher CPU loads (e.g., 75%, 85%, and 95%) for the *inline* policy which may result in higher accuracy than expected. (The claim's main focus is the *join* experiments.)

Optional (E-R3): increase # of samples. By default, `unfilter-rls claims run C-R3` only takes 10k samples for each cell. In the paper, we use 100k samples (which tightens the confidence intervals). You can change number of samples by overriding the default `table1.probes` option:

```
unfilter-rls claims run C-R3 \
  --set table1.probes 100000
# Warning: this will 10x the
# compute-hour estimate
```

(E-R4): Cross-region latency vs accuracy (Table 2) *[12 compute-hours]*
This experiment performs the cross-region experiments in Table 2, with an additional factor of CPU load introduced by a 3rd *noise* VM.
Results: Results are written to:

```
rls/results/table2/table2.pdf
```

Compare this to **Table 2** in the paper.

Optional (E-R4): increase # of samples. By default, `unfilter-rls claims run C-R4` only takes 1k samples for each cell. In the paper, we use 10k samples (which tightens the confidence intervals).[a] You can change number of samples by overriding the default `crosszone.probes` option:

```
unfilter-rls claims run C-R4 \
  --set crosszone.probes 10000
# Warning: this will 10x the
# compute-hour estimate
```

[a] The number of samples for the cross-zone experiments is 10× less than the same-zone experiments as the round-trip time is about 10× higher due to the network latency.

(E-R5): Transparent data encryption (no figures) *[10 compute-minutes]*
This experiment validates that TDE does not change the efficacy of the timing attack. It is in a separate script as it requires reprovisioning the VMs to install LUKS.
Results: Results are written to:

```
rls/results/existence/
existence_tde_figure.pgf
rls/results/existence/
existence_tde_figure.tex
```

Verify that the two figures look visually similar to **Figure 2** and **Figure 3** in the paper.

(E-R6): Binary search / parallelism for attribute domain recovery (Table 3) *[8 compute-hours]*
This validates that the binary search attack (with optional parallelism) outperforms the prior linear attacks on large, sparse domains. It shows it performs slightly worse on dense domains. It also shows that too much parallelism can lower the accuracy due to self-imposed CPU load on the database.
Results: Results are written as a LaTeX file to:

```
rls/results/table3/table3.tex
```

Compare this to **Table 3** in the paper. For the higher parallelism experiments (workers > 4), the additional variance and binary search nature of the attack means that accuracy may be occasionally lower than the averages reported in the paper (e.g. if the attack got unlucky at a higher tree level and mistakenly pruned much of the search space). (This is why we run each row 10 times in the paper.)

Optional (E-R6): increase # of repetitions. By default, `unfilter-rls claims run C-R6` runs each experiment row 1 time. In the paper, we use 10 repetitions (which tightens the confidence intervals). You can change number of repetitions by overriding the default `singleattr.reps` option:

```
unfilter-rls claims run C-R6 \
  --set singleattr.reps 10
# Warning: this will 10x the
# compute-hour estimate
```

(E-R7): Selectivity ordering vs tuple reconstruction accuracy (Table 4) *[12 compute-hours]*
This experiment validates our theoretical analysis of how the accuracy / performance of the attack changes based on the order of the attributes used in the tuple reconstruction algorithm.
Results: Results are written as a LaTeX file to:

```
rls/results/table4/table4.tex
```

Compare this to **Table 4** in the paper. For the reconstruction orderings starting with `zip` and `age`, you sometimes will get significantly higher *Recall* and *Precision* than what is in the paper due to inherent

variance in the binary search and oracle accuracy, but this should come at the cost of significantly higher *Queries* and *Time* because the lack of accuracy results in more false positives, resulting in unnecessary trie traversals. (This is why we run each row 3 times in the paper.) The most important factor to check for is that *ssn*-leading orderings should outperform the *zip*- or *age*-leading orderings in either Recall / Precision, or Queries / Time.

Optional (E-R7): increase # of repetitions. By default, `unfilter-rls claims run C-R7` runs each experiment row 1 time. In the paper, we use 3 repetitions (which tightens the confidence intervals). You can change the number of repetitions by overriding the default `tuplext.reps` option:

```
unfilter-rls claims run C-R7 \
  --set tuplext.reps 3
# Warning: this will 3x the
# compute-hour estimate
```

(E-R8): Mitigation ablation experiments (Table 5) *[5 compute-minutes]*
This experiment validates our analysis of the necessary components needed to mitigate the attack.
Results: Results are written to:

```
rls/results/table5/table5.pdf
```

Compare this to **Table 5** in the paper.

(E-R9): Dataset statistics ("Schema and data" paragraph) *[1 compute-minute]*
This experiment validates some summary statistics about the datasets used in the experiments.
Results: Results are written to:

```
rls/results/dbsize/summary.txt
rls/results/dbsize/db_sizes.json
```

Verify the statistics align with the **"Schema and data" paragraph in Section 3.3** of the paper.

(E-D_): Run all DLS experiments *[5 human-minutes + 16–18 compute-hours + 64GB disk + 60GB memory]*
For evaluator convenience, we provide a script which runs all of the **DLS** experiments in a single pipeline.
Execution: We provide a one-shot script which runs all of the DLS experiments in sequence. This command runs the full experiment pipeline,[1] which will take 16–18 compute-hours:

```
bash dls/run.sh
```

[1] For convenience, the script will cache results for dataset runs after they have successfully completed (e.g., in case you accidentally stop the pipeline). If you would like to rerun existing results, add the `--destructive-rerun-existing-results` flag.

Optional (E-D_): shorter / less memory experiments. To reduce time to 2–3 compute-hours and the required RAM (< 16 GB RAM), you can skip the longer $|D| = 1000$ run by only picking the $|D| \in \{1, 10, 100\}$ datasets:

```
bash dls/run.sh --datasets 1,10,100
```

Optional (E-D_): increase # of reps. By default, the `run.sh` script runs 1 run of each attack. In the paper, we ran each attack 3 times to take wall-clock time averages. You can specify the # of reps using the `--trials` flag:

```
bash dls/run.sh --trials 3
# Warning: this will 3x the compute-
hour estimate
```

This script runs the following experiments:

(E-D1): Run attack on OpenSearch (Table 6)
This experiment verifies that the DLS attacks recover the indexed n-gram set exactly across all tested corpus sizes ($|D| \in \{1, 10, 100, 1000\}$) on OpenSearch 3.6.0.
Results: Results are aggregated and written to

```
dls/results/reviewer/opensearch_table.tex
```

Verify that all columns are similar to **Table 6** in the paper. In particular, "Logical Queries" should be identical.

Optional (E-D1): raw data. If you want to see the raw statistics (not yet rendered into TEX), see the `dls/results/reviewer/stats` folder. If you want to see the full attack logging output, see the `dls/results/reviewer/logs` folder.

(E-D2): Run attack on Elasticsearch (Table 6)
This experiment verifies that the DLS attacks recover the indexed n-gram set exactly across all tested corpus sizes ($|D| \in \{1, 10, 100, 1000\}$) on Elasticsearch 9.3.3.
Results: Results are aggregated and written to

```
dls/results/reviewer/elasticsearch_table.
tex
```

Verify that all columns are similar to **Table 6** in the paper. In particular, "Logical Queries" should be identical.

Optional (E-D2): raw data. If you want to see the raw statistics (not yet rendered into TEX), see the `dls/results/reviewer/stats` folder. If you want to see the full attack logging output, see the `dls/results/reviewer/logs` folder.

(**E-D3**): **Validate de Bruijn reconstruction produces "reasonable" results (Figure 5)**
This experiment demonstrates the ability of the greedy de Bruijn reconstruction to reconstruct "reasonable" approximate document text. It specifically operates against the recovered 4-grams from the individual dataset runs done as part of (**E-D1**) and (**E-D2**).
Results: Results are written to

```
dls/results/reviewer/reconstruction/
opensearch-d10-r1_debruijn_greedy.txt
```

Verify that `contig_2` and `contig_6` match the contents of **Figure 5**. If interested, you can view other reconstructions from the $|D| = 10$ attacks in the same file, or view reconstructions from other attacks in the other files in `dls/results/reviewer/reconstruction`.

A.5 Troubleshooting Guide

- **RLS** **`ZONE_RESOURCE_POOL_EXHAUSTED` or `STOCKOUT`:** This error occurs when Google Cloud is out of `c4-standard-*` VMs in a particular zone (likely the default zone of `us-central1-a` used in the experiments). You can change the zone by the `--set zone` configuration option. For example:

```
unfilter-rls claims run C-R9 \
  --set zone us-east1-b
```

- **DLS** **`UNEXPECTED_EOF_WHILE_READING`:** This error is usually caused by the database server crashing due to an out-of-memory (OOM) error. Make sure that your system meets the minimum memory requirements specified for the experiment.
- **DLS** **`index_create_block_exception`:** This error is usually caused by the database server preventing additional uploads due to insufficient disk space. Make sure that your system meets the minimum storage requirements specified for the experiment.

A.6 Notes on Reusability

For **RLS**:

- Please see the "Bring Your Own Machines" in the `README.md` if you would like to run the **RLS** experiments on different (non-GCE) machines. It involves creating a new configuration file that points to your chosen machines.

For **DLS**:

- `dls/setup.sh`, `dls/run.sh`, and `dls/teardown.sh` helper scripts have an optional `--help` flag which provides additional options for running the scripts.
- `dls/run.sh` invokes the `python -m enumerate` module, which itself has several command-line options for changing the datasets to attack or different attack options. For more details, see the documentation in `dls/enumerate/README.md`.
- `dls/dataset/util/parse_hf_dataset.py` can be used to generate more dataset files from other Hugging Face datasets for testing the attack on other datasets. See the `dls/dataset/README.md` for more details.
- `dls/config/config.yml` contains an `attack` block which controls parameters used in the experiments. For example, the `chars` field controls the set of characters used to explore the term trie. You can change the `chars` field to see how that affects the runtime of the attack.

A.7 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: PICASSO: Scaling CHERI Use-After-Free Protection to Millions of Allocations using Colored Capabilities

Merve Gülmez*, Ruben Sturm†‡, Hossam ElAtali§, Håkan Englund*,
Jonathan Woodruff‖, N. Asokan§,¶, Thomas Nyman#
*Ericsson Security Research, †DistriNet, KU Leuven, §University of Waterloo,
‖University of Cambridge, ¶KTH Royal Institute of Technology, #Ericsson Product Security
{merve.gulmez,hakan.englund,thomas.nyman}@ericsson.com, ruben.sturm@kuleuven.be
hossam.elatali@uwaterloo.ca, jonathan.woodruff@cl.cam.ac.uk, asokan@acm.org

A Artifact Appendix

This paper introduces *colored capabilities* that add a controlled form of indirection to CHERI's capability model. This enables *provenance tracking* of capabilities to their respective allocations via a hardware-managed *provenance-validity table*, allowing bulk retraction of dangling pointers without needing to quarantine freed memory. Colored capabilities significantly reduce the frequency of capability revocation sweeps while improving security. Colored capabilities are realized in PICASSO, an extension of the CHERI-RISC-V architecture on a speculative out-of-order FPGA softcore (CHERI-Toooba). Colored-capability support is integrated into the CheriBSD OS and CHERI-enabled Clang/LLVM toolchain. PICASSO successfully detects use-after-free and double-free defects in NIST Juliet test cases without false positives and real-world CVEs in BZip2 (CVE-2016-3189), libiberty (CVE-2016-4487), nasm (CVE-2017-10686, CVE-2019-8343), libzip (CVE-2019-17582), NGINX (CVE-2020-24346), LibreDWG (CVE-2022-35164), Lua (CVE-2019-6706), and three other use-after-free issues reported in mjs (issues 73 and 78) and yasm (issue 91). The efficiency of PICASSO is demonstrated using the SPEC CPU benchmark and real-world SQLite, PostgreSQL, and gRPC benchmarks.

A.1 Abstract

This artifact provides the complete implementation of PICASSO on the speculative out-of-order CHERI-Toooba FPGA softcore, along with the Colored capability-enabled CheriBSD operating system, Clang/LLVM toolchain, QEMU implementation, runtime setup, and detailed instructions necessary to reproduce the experimental results.

The artifact supports three evaluation environments:

- *A QEMU-based emulation environment* with support for colored-capability-enabled CheriBSD. This environment can be used to reproduce the security evaluation using the NIST Juliet test suite, real-world CVEs, and the other real-world use-after-free issues.
- *A bare-metal hardware simulation environment* based on the Bluespec simulator. This environment can be used to reproduce performance evaluation using the MiBench suite and Coremark.
- *An FPGA-based evaluation environment* with support for colored-capability-enabled CheriBSD. This environment can be used to reproduce the security evaluation and the performance evaluation using the CheriBSD-based benchmarks (SPEC, SQLite, PostgreSQL, and gRPC).

> **Note:** The Artifact Evaluation Committee evaluated the artifact in the QEMU emulation and bare-metal hardware simulation environments. Further details on the reproduced experiments are provided in § A.4.2.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Theoretically methods such as PICASSO can be misused for malicious vulnerability discovery. However, executing PICASSO on the field-programmable gate array (FPGA) platform or QEMU-based emulation environment supported by our artifact is orders of magnitude slower than alternative software-based approaches for vulnerability detection, such as sanitizer-instrumented binaries. Consequently, the PICASSO artifact does not provide additional capabilities beyond such existing tools. We therefore do not expect security, privacy, or ethical concerns associated with the PICASSO artifact.

[3] R.S. contributed to the research while at Ericsson Security Research.

A.2.2 How to access

Our artifact can be accessed at: https://github.com/coloredcapabilities/colored-artifact and https://doi.org/10.5281/zenodo.20353117.

A.2.3 Hardware dependencies

The QEMU-based emulation environment, the bare-metal hardware simulation environment, and the build environment for software artifacts targeting each evaluation environment require a **x86_64 Linux machine** with Docker installed. The build for the docker images requires roughly 120 GB of disk space.

The FPGA-based evaluation environment requires a **Xilinx/AMD VCU118 (UltraScale+ Virtex) evaluation board**, and a host machine with JTAG/Vivado Hardware Manager.

A.2.4 Software dependencies

The artifact provides a Dockerfile for the QEMU-based emulation environment, the bare-metal hardware simulation environment, and the build environment for software artifacts. The Dockerfile has been tested on **Ubuntu Server 24.04.4 LTS** using **Docker version 29.1.3** obtained from https://ubuntu.com/server

Xilinx/AMD Vivado Design Suite 2019.1 and license is required to build/flash bitstreams for the FPGA-based evaluation environment. Pre-built bitstreams are also provided, requiring only a means of programming the board, e.g. JTAG/Vivado Hardware Manager from **Vivado Lab 2019.1**. The Vivado Design Suite and Lab 2019.1 installers can be obtained from https://www.amd.com/en/support/downloads/adaptive-socs-and-fpgas/development-tools/2019-1.html

Reproducing the performance evaluation using SPEC CPU requires the **SPEC 2006 CPU ISO image** and license from https://www.spec.org/cpu2006/

A.2.5 Benchmarks

We use the NIST Juliet test suite, eight known CVEs and three other use-after-free issues for the security evaluation of `PICASSO` and Mibench, Coremark, SPEC SPEC CPU 2006, SQLite, PostgreSQL, and gRPC benchmarks to evaluate the performance.

A.3 Set-up

Before installing the `PICASSO` artifact for the first time, please perform a clean install of **Ubuntu Server 24.04.4 LTS**, install **Docker**, and add `$USER` to the `docker` group:

```
sudo apt install docker.io
sudo usermod -aG docker $USER
```

Logout, then login again for the group change to take effect.

On the host connected to the VCU118 evaluation board, download and install **Vivado Design Suite 2019.1**. A license key for the tool should have been included with your VCU118 board. See the Vivado Design Suite User Guide for installation instructions: https://docs.amd.com/v/u/2019.1-English/ug973-vivado-release-notes-install-license

If using different machines for building the artifact and FPGA host, only the build machine needs a license in order to build new bitstreams. In this case you can install **Vivado Lab 2019.1** (obtained from the same URL as Vivado Design Suite) on the FPGA host machine because it does not require a license and is solely used to program the FPGA.

A.3.1 Installation

1. Clone the artifact repository:

```
git clone
 ↪  https://github.com/coloredcapabilities/colored-artifact.git
```

2. Build the `PICASSO` container image:

```
cd colored-artifact
## 112 GiB, and takes 7 hours, it builds the whole
 ↪  stack, qemu, bluespec simulator, llvm, cheribsd,
 ↪  gdb
docker build --network=host -f Dockerfile -t picasso .
```

A.3.2 Basic Test

```
# Start the container
docker run -i -t  --name picasso-run picasso

# Boot CheriBSD
cd ~/cheri/cheribuild
./cheribuild.py
run-riscv64-purecap --skip-update
 ↪  --run-riscv64-purecap/ssh-forwarding-port 10222
```

The basic test should result in a CheriBSD login prompt. Entering `root` as the username (no password is needed) should result in a funtional shell in the CheriBSD system.

The artifact directory contains a `FPGA.md` file with instructions for flashing the FPGA with CheriBSD and obtaining a shell on the FPGA guest system.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): `PICASSO` detect all use-after-free and double-free violations in the heap-based temporal-memory-safety CWEs of the NIST Juliet test suite (CWE-415/CWE-416) and in eight real-world use-after-free/double-free CVEs and three other use-after-free issues. This is demonstrated by experiment (E1).

(C2): The cycle-count overhead of PICASSO on the MiBench suite and Coremark running on the Bluespec simulator is small. This is demonstrated in experiment (E2), corresponding to Table 3 and Table 5 in Appendix C in the paper.

(C3): PICASSO incurs only a small (≈5% geometric mean) execution-time overhead over baseline CHERI-Toooba on SPEC CPU2006, while requiring substantially less frequent capability revocation sweeps than prior sweeping-based approaches. This is proven by experiment (E3), corresponding to §7.2 and Figure 10 in the paper.

(C4): For long-running workloads (SQLite, PostgreSQL, gRPC), PICASSO provides lower and more consistent latency than prior CHERI temporal-safety mechanisms that rely on periodic memory-sweeping revocation. This is proven by experiment (E4, E5, E6), corresponding to §7.2 and Figures 11, 12, and 13 in the paper.

A.4.2 Experiments

> **Note:** E1, E2, and E4 were evaluated and successfully reproduced by the Artifact Evaluation Committee. E3, E5, and E6 require specific hardware and software licenses. Due to these hardware and license dependencies, E3, E5, and E6 could not be reproduced during the artifact evaluation.

(E1): *Security evaluation (Juliet & real-world CVEs)* [1 human-hour + 1–2 compute-hours]: demonstrates that PICASSO detects all heap UAF/double-free violations in Juliet CWE-415/CWE-416, eight real-world CVEs, and three other use-after-free bugs. Can be run on QEMU or on the FPGA.

Preparation: The Juliet test suite, CVE and bug PoCs are pre-built in the container image.

```
### Start the container
docker run -i -t  --name picasso-run picasso

## Boot CheriBSD in Terminal 1
cd cheribuild
./cheribuild.py run-riscv64-purecap --skip-update
 ↪  --run-riscv64-purecap/ssh-forwarding-port 10222

## Open Terminal 2 and run:
## Transfer Juliet binaries
docker exec -it picasso-run bash
~/cheri/utils_script/transfer_juliet.sh
 ↪  root@127.0.0.1 -p 10222

## Run in Terminal 2:
## Transfer CVE and bug tests
~/cheri/validation/transfer.sh root@127.0.0.1 -p
 ↪  10222
```

Execution: To run Juliet test suite, CVE and bug PoCs:

```
# Run in Terminal 1:
cd /root/juliet-bin
sh juliet-run.sh 415 2s < /tmp/in.txt
sh juliet-run.sh 416 2s < /tmp/in.txt

## Run in Terminal 2:
## Run the CVE and bug tests
cd /home/ubuntu/cheri
validation/run_all_remote.sh root@127.0.0.1 -p
 ↪  10222
## Collect juliet result
utils_script/collect_juliet_results.sh
 ↪  root@127.0.0.1 -p 10222
```

Results: For CWE-415 (Double Free), the program exits with code 255 (`exit(-1)`). For CWE-416 (Use After Free), the program receives signal 34 (SIGPROT, an in-address space CHERI capability exception), resulting in exit code 162. For each of the CVEs, and issue reports `DETECTED (signal 34)` for UAF cases or exit code 255 for double-free cases; an exit code of 0 (`VULNERABLE`) would indicate the bug was not caught.

(E2): *PICASSO run-time overhead (MiBench and Coremark)* [15 human-minutes + 1 compute-hours for MiBench – 8 compute-minutes for Coremark]:

Preparation: The Bluespec simulator, Mibench, and Coremark binaries are pre-built in the container image.

```
### Start the container (if not already running)
docker run -i -t  --name picasso-run picasso
```

Execution: Inside the Docker images

```
## Run in shell inside container:
cd /home/ubuntu/bench
./run_mibench.sh

## Run in shell inside container:
cd /home/ubuntu/bench/coremark
./run_coremark_for_sim.sh
```

Results: MiBench incurred an average overhead of 1.32% in simulation. For CoreMark, PICASSO purecap incurred overheads of 3.12% and 6.14%, depending on the configuration. The simulated CoreMark overheads differ slightly from the FPGA results reported in the paper (§7.2).

(E3): *PICASSO run-time overhead (SPEC CPU2006)* [1 human-hour + 42 compute-hours]: Due to licensing restrictions, we do not provide the SPEC CPU2006 source code. Evaluators must have their own valid SPEC CPU2006 license to reproduce these results.

Preparation: To build SPEC for CheriBSD:

```
### Start the container in Terminal 1
### (if not already running)
docker run -i -t --name picasso-run picasso

## Transfer SPEC CPU2006 ISO from Terminal 2:
docker cp /path/to/cpu2006-1.2.iso
 ↪  picasso-run:/home/ubuntu/
```

```
## Run in Terminal 1
cd cheribuild
./cheribuild.py spec2006-riscv64-purecap
↪ --spec2006/iso-path /home/ubuntu/cpu2006-1.2.iso
```

The built SPEC folder is placed at:

```
~/cheri/build/spec2006-riscv64-purecap-build
```

Copy the built SPEC folder to the **host** system:

```
## Run in Terminal 2:
mkdir -p ~/cheri/build
docker cp
↪ picasso-run:~/cheri/build/spec2006-riscv64-purecap-build
↪ ~/cheri/build/
```

Once copied, built SPEC folder on the **host** is placed at:

```
~/cheri/build/spec2006-riscv64-purecap-build
```

Now prepare the SPEC benchmark folder from a shell on the **host**:

```
# Remove unnecessary files to reduce transfer size
$~/cheri/utils_script/spec/spec_folder_reduce_folder.sh

# Instrument benchmark run scripts with timing/stat counters
python3 ~/cheri/utils_script/spec/automate.py
```

Copy the SPEC folder and `$ARTIFACT_DIR/utils_script/spec/run_all_spec_fpga.sh` to the FPGA (see `FPGA.md` included in artifact for copying instructions).

Execution: Run inside **FPGA guest**:

```
cd /bench/SPEC/CINT2006
sh /path/to/run_all_spec_fpga.sh
↪ /bench/SPEC/CINT2006
```

Results: The results for each benchmark are output to:

```
<benchmark>/<benchmark>_OUTPUT/
```

After collecting results for all each benchmark, place the resulting `_OUTPUT` directories under `utils_script/spec/colored_paper/` directory:

```
utils_script/spec/colored\_paper/<benchmark>\_OUTPUT/
```

We include the output directories for our experiments using all configurations (`baseline`, `colored_paper`, `cornucupia`) in the artifact. Replace the `colored_paper` output directories to obtain the overhead tables for the reproduced results. Once the reproduced output directories are in place, run the analysis script to generate the overhead tables and figures:

```
python3
↪ $ARTIFACT_DIR/utils_script/spec/analyze_spec_overhead.py
```

This produces:

- Per-benchmark cycle, memory, tagcache, and DRAM traffic overhead tables
- Geometric mean summaries across all benchmarks
- Overhead figures saved as PDF and PNG alongside the script

Comparing the resulting tables with the reference data in the paper should yield comparable results.

(E4): *PICASSO memory overhead and reduce revocation sweep (SQLite) for QEMU* [5 human minutes – 30 compute-minutes for QEMU, 1 hours for SQLite]:
Preparation: SQLite binaries are pre-built in the container image.

```
### Start the container (if not already running)
docker run -i -t --name picasso-run picasso

### Run PICASSO QEMU
./cheribuild.py run-riscv64-purecap --skip-update
↪ --run-riscv64-purecap/ssh-forwarding-port 10222

### Run Baseline QEMU in the third terminal
~/cheri/utils_script/run_baseline_qemu.sh
```

Execution: Inside the container

```
## Run in shell inside container:
cd ~/cheri/utils_script/sqlite
## For PICASSO
./run_speedtest1.sh root@127.0.0.1 -p 10222

## For Baseline
./run_speedtest1.sh root@127.0.0.1 -p 10223

## Check the results, after the speedtest1 results,
↪ look at the mrs[process_id] revoke counter
cat ./speedtest1_logs/speedtest1_$date.log
```

Results: Compared to the baseline, the results show improved performance with zero PICASSO revoke events (vs. 272 in the baseline) and a significantly lower maximum resident set size of 13,268 (vs. 30,708), indicating reduced memory usage and better resource efficiency.

(E5): *PICASSO memory overhead and reduce revocation sweep (PostgreSQL) for FPGA* [30 human minutes – 3 hours for FPGA]:
Preparation: PostgreSQL binaries are not pre-built in the container image. Please compile from scratch.

```
### Start the container (if not already running)
docker run -i -t --name picasso-run picasso
### Compile the postgres
cd $CHERI_ROOT
git clone
↪ https://github.com/CTSRD-CHERI/postgres.git
cd postgres
git apply $ARTIFACT_DIR/patches/postgress.diff
cd $CHERIBUILD
./cheribuild.py postgres-riscv64-purecap -d
```

Execution: Copy scripts to FPGA

```
## Inside the FPGA
## creates the database and runs pgbench - 60
↪ minutes
sh ./postgres-bench-stats.sh

## Inside the Host
```

```
cd ~/cheri/utils_script
sh postgress/pgbench-client.sh

postgress/plot_combined_cdf.py
```

Results: This produces the Figure 13 in the paper. For p50 latency, PICASSO incurs a 3% overhead, while Cornucopia incurs a 7% overhead. The raw output can be found at `/cheri/utils_script/postgress`

(E6): *PICASSO memory overhead and reduce revocation sweep gRPS for FPGA* [30 human minutes – 4 hours for FPGA]:
Preparation: Build inside the container

```
### Start the container (if not already running)
docker run -i -t  --name picasso-run picasso
### Compile the qrps
cd ~/cheri/cheribuild

./cheribuild.py grpc-native -d
./cheribuild.py grpc-riscv64-purecap -d
```

Execution: Run the in the host machine

```
~/cheri/utils_script/grpc/grpc-client-bytes.sh

## get the plot
~/cheri/utils_script/grpc/analyze_qps_boxplot.py
```

Results: PICASSO degrades transactions per second (TPS) by 6.2%, compared to 23.1% for Cornucopia. Figure 13 plots individual transaction latencies, Cornucopia exhibits periodic latency spikes occurring approximately every 20 transactions (60 allocations per transaction).

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Trust Nothing: RTOS Security without Run-Time Software TCB

Eric Ackermann, Sven Bugiel
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

Embedded devices face an ever-expanding threat landscape: vulnerabilities in application software, operating system kernels, and peripherals threaten the embedded device integrity. Existing computer-architectural defenses fully consider at most two of these threat vectors in their security model.

Our paper aims at addressing this gap using a novel capability architecture. To this end, we combine a token capability approach suitable for building an untrusted operating system with protection against malicious devices without requiring hardware changes to peripherals.

First, we develop and evaluate Bredi, a full FPGA implementation of our capability architecture around legacy hardware components. Further, we present a soft real-time operating system Skadi based on Zephyr that has no run-time software TCB. To this end, we disaggregate Zephyr's subsystems into small, mutually isolated components. All subsystems that exist at run time, including scheduler, allocator and DMA drivers, and all peripherals are fully untrusted. We believe that our work offers a foundation for more rigorous security-by-design in tomorrow's security-critical embedded devices.

We evaluate the overhead of our design. While overheads are significant, we manage to demonstrate that our system can be used in practical scenarios, and room for improvement continues to exist.

Our artifact comprises Skadi, Bredi, instructions for configuring and running the systems as well as raw data from our benchmark executions.

A.2 Description & Requirements

Requirements for building the FPGA SoC. Bredi targets the *Digilent Genesys2* FPGA board. Building the SoC for this board requires *Xilinx Vivado version 2024.2* with a license appropriate for the xc7k325tffg900-2 part (evaluation license should be sufficient). Furthermore, we utilize Xilinx' proprietary AXI ethernet subsystem. To this end, building the SoC requires the appropriate licenses (EF-DI-TEMAC-PROJ or EF-DI-TEMAC-SITE, evaluation license suffices). Provided scripts assume an *Ubuntu 22.04 host system* with root access. Furthermore, for accessing area reports and troubleshooting, we require GUI access to Vivado (e.g., via X forwarding).

It is possible to evaluate the *functionality* of our artifact (sans the DMA) on the Digilent Arty A7-100t FPGA board. This board requires *only free (built-in) Vivado licenses*.

Requirements for building Skadi RTOS. We provide scripts for building Skadi from source. This requires a workstation with Ubuntu 22.04, root access and sufficient free disk space (>32 GB).

Requirements for running the Artifact. All of our evaluations were conducted on a *Digilent Genesys2* FPGA board. Furthermore, we require an SD card, (optionally, for the Arty A7) a suitable USB JTAG debugger (we use a *Digilent HS2*), jumper cables, micro USB cables and an Ethernet connection to a workstation for network performance testing. We used a Raspberry Pi 4 for network performance testing.

Included benchmarks.

The artifact comprises the following benchmarks relevant for our performance claims:

- FPGA chip area can be reported by Vivado in GUI mode (table 2 in the paper).
- ASIC chip area can be reported using our provided scripts (table 3 in the paper). As the run time for the entire ASIC pipeline exceeds the AEC timeframe, and ASIC area is not a major claim of our work, we have elected to *not* reproduce it here. If you want to reproduce these results, follow the instructions under *7.1 - Chip Area—ASIC* in our artifact README and plan with 1-2 weeks of run time.
- Code size is reported using *GNU size* (baseline) and via the console (Skadi) for selected samples (table 4 in the paper).
- Load time is reported via the console for selected samples (table 5 in the paper).
- We provide a custom microbenchmark *sim_capability_test* (table 6, 7 in the paper).
- We use an existing microbenchmark from Zephyr *syscall_perf* (table 7 in the paper).

- We ship the *coremark* and the *stream* benchmarks in a ported version for measuring compute performance (table 8 in the paper).
- For network benchmarks, we use the *ping* and *iperf* utilities with a ported *zperf* sample, a custom shell script and sample for measuring RX/TX latency and Eclipse Mosquitto for measuring MQTT-over-TLS performance (table 9 in the paper).
- Finally, for IRQ performance, we use a custom sample in two modes (table 10 in the paper).

A.2.1 Security, privacy, and ethical concerns

Skadi and Bredi are purely defensive in nature (see ethical discussion in our paper). We do not anticipate any security or privacy risks from using the artifact.

A.2.2 How to access

The permanent reference for the version of the artifact used for our original benchmarks is https://doi.org/10.5281/zenodo.20259754. The version of the artifact that was evaluated during the AE process is https://doi.org/10.5281/zenodo.21486644. An active development version can be found at https://github.com/cispa/Northcape. *Please note:* The updated versions have additional security and performance improvements beyond the original artifact, so there will be some deviation in the results. We aim for the same *order of magnitute* as the results reported in the paper.

For reproducing the results found in the paper, please use the original version of the artifact.

A.2.3 Hardware dependencies

For artifact evaluation, we provided remote access to a workstation suitable for building and running the artifact, including a fully configured Genesys2 FPGA board. For reproducing our results independently, a Digilent Genesys 2 FPGA board and the accessories mentioned above are needed. For building the hardware and software, we recommend a workstation with at least 32 GiB RAM and 200 GB of available disk space.

A.2.4 Software dependencies

Our scripts assume Ubuntu 22.04 with docker installed. In order to install Xilinx Vivado 2024.2, please follow the official instructions to register with Xilinx and download version 2024.2 of the software: https://www.xilinx.com/support/download/index.html/content/xilinx/en/downloadNav/vivado-design-tools/2024-2.html. Make sure to install *Vitis* (including Vivado) and ensure you install board files for at least *Kintex 7*.

A.2.5 Benchmarks

All benchmarks are provided in source code form with the artifact.

A.3 Set-up

A.3.1 Installation

Please make sure that Vivado version 2024.2 is installed first. Our scripts assume it is in */opt/Xilinx/Vivado/2024.2/*.

Then, from the top directory, run our scripts for compiling the FPGA bitstream (hardware):

```
cd scripts/Vivado
# Bredi SoC
./create_project.sh
./create_bitstream.sh
# Baseline for SoC without Bredi
./create_project_non_bredi.sh
./create_bitstream_non_bredi.sh
```

FPGA synthesis is non-interactive and should take around 2-3 hours.

In parallel to or after compiling the hardware, you can compile the *Linux* and *Zephyr* baselines:

```
# Linux
sudo apt-get update &&
sudo apt-get install -y \
        autoconf \
        automake \
        autotools-dev \
        curl \
        libmpc-dev \
        libmpfr-dev \
        libgmp-dev \
        libusb-1.0-0-dev \
        gawk \
        build-essential \
        bison \
        flex \
        texinfo \
        gperf \
        libtool \
        patchutils \
        bc \
        zlib1g-dev \
        device-tree-compiler \
        pkg-config \
        libexpat-dev \
        wget \
        cpio \
        python3 \
        python3-pip \
        unzip \
        rsync \
        git \
        picocom \
        psmisc
```

```
cd software/include/northcape-linux-stack
# Linux setup
make images
# flash SD card.
# ALL DATA ON SDDEVICE ARE LOST
sudo make flash-sdcard SDDEVICE=/dev/sdX
chmod 777 scripts/ssh*
# Zephyr / Skadi setup
cd ../northcape-zephyr-stack/
sudo docker build -t skadi_build:latest . \
-f scripts/skadi/Dockerfile
cd ../
# adapt USB device: you can get bus
# and function number XXX and YYY
# from lsusb
# look for ID 0403:6010, FT2232
sudo docker run -v .:/workspace \
--device /dev/bus/usb/XXX/YYY \
-it skadi_build:latest bash
# within the container
cd northcape-zephyr-stack
./scripts/skadi/install-dependencies.sh
```

Software compilation involves building a GNU toolchain from scratch with a custom configuration. In combination, Linux and Zephyr can compile for 5 or more hours. Also, Linux and Zephyr/Skadi setup can be run in parallel.

The provided scripts configure a Docker container for compiling Zephyr/Skadi. *Note:* You can use the same scripts to configure a persistent build environment. However, this will modify your home and temp directory.

A.3.2 Basic Test

We suggest two distinct basic tests: Booting Linux into a root shell and loading Skadi's hello world sample.

Booting Linux. Insert the SD card into the board. From the top directory, run the following commands to launch Vivado interactively and run the hardware server:

```
cd scripts/Vivado
# must have GUI environment
# e.g., SSH X forwarding
./open_project.sh
# second shell, in scripts/Vivado
./launch_hw_server.sh
# third shell
# adapt to Genesys' FTDI UART
# add your user to dialout
# or run as root
picocom -b 115200 /dev/serial/by-id/*
```

Start the TFTP server in a separate shell:

```
cd software/include/
cd northcape-linux-stack/scripts
./start-tftp.sh
```

When Vivado has fully loaded the project, expand *Open Hardware Manager* in the left-most *Flow Navigator* pane. Click *Open Target* and *Open New Target*. Select *Remote server* and configure its location as `localhost:3121`. The dialog should list an *xck7* device, corresponding to the Genesys 2. Accept defaults and continue through the dialog.

You can now click *Program Device* and the *xck7* device in the *Flow Navigator*. The bit stream generated in the setup should be pre-filled in the dialog, and the probes file is optional. Hit *Program* to program the device.

After programming the FPGA, you should receive output from the boot ROM, OpenSBI and u-boot stages without further action. If everything works correctly, the board should boot into a root shell.

Running Skadi's hello-world sample.

Build and load the hello-world sample:

```
cd software/include
# mount a new container if needed
sudo docker run -v .:/workspace \
--device /dev/bus/usb/XXX/YYY \
-it skadi_build:latest bash
cd northcape-zephyr-stack
# run this in every new Docker container
# already run in the first container
./scripts/skadi/setup-docker.sh
source .venv/bin/activate
# build the sample
west build -p always -b \
    cv64a6_genesysII \
    samples/hello_world
west debug --skip-rebuild
```

When GDB has completed loading the sample, hit *c* and enter to continue. You should see Skadi relocating its subsystems one by one before printing a final hello world message.

Please note previous versions of the artifact were prone to errors in the debug module, causing loading Skadi/Zephyr to fail. We provide a fix of this problem in the final version of the artifact.

Loading the hello-world sample should take less than one minute. You should see GDB load the initial sections of the loader ELF relatively quickly, followed by a longer transfer of the read-only data section. If you receive an error from GDB or it does not start loading any sections, repeat *west debug*. If the second attempt fails as well, kill all instances of *openocd*, program the FPGA again via Vivado and try again.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Bredi's area overhead matches the values given in table 2 of the paper, with small variations. This is proven by experiment (E1).

(C2): Bredi's code size overhead matches the values given in table 4 of the paper, with small variations. This is proven by experiment (E2).

(C3): Bredi's load time overhead matches the values given in table 5 of the paper, with small variations. This is proven by experiment (E3).

(C4): Bredi's operation microbenchmarks are fast, consuming less than 1000 cycles from a software and less than 100 cycles from a hardware perspective each, matching the values in table 6 and 7. This is proven by experiment (E4).

(C5): Bredi's compute performance is indistinguishable from the Zephyr baseline and faster than the Linux baseline, matching table 8. This is proven by experiment (E5).

(C6): Bredi's network performance overhead over Zephyr is less than factor three for the ping, TCP performance and TX duration benchmarks, matching table 9 from the paper. This is proven by experiment (E6).

(C7): Bredi's network performance overhead over Zephyr is approximately 4 for the RX duration benchmark, matching table 9 from the paper. This is proven by experiment (E6).

(C8): Bredi's MQTT performance overhead over Zephyr is less than 10%, matching table 9. This is proven by experiment (E6).

(C9): Bredi's network performance overhead over Linux is less than 2 for the ping, RX and TX duration benchmarks, matching table 9 from the paper. This is proven by experiment (E6).

(C10): Bredi's IRQ performance overhead over Zephyr is less than 3 for independent interrupts. This is proven by (E7).

(C11): Bredi's IRQ performance overhead over Zephyr is less than 10 for monotonic interrupts. This is proven by (E7).

A.4.2 Experiments

(E1): [Chip Area] [5 human-minutes]: Determine the FPGA Area overhead of Bredi.
Preparation: Synthesize both versions of the SoC as explained in the Set-up steps.
Execution: Follow the steps under *7.1 - Chip Area—Northcape FPGA SoC* in *paper_artifact/README.md*.
Results: The results comprise the chip area reported by Vivado for each component.

(E2): [Code Size] [30 human-minutes]: Determine the code size of Skadi compared to its baseline, Zephyr.
Preparation: Synthesize and program the FPGA configuration. Setup the software toolchain.
Execution: Follow the steps under *7.2 - Software: Code Size—zephyr Code Size - Genesys2* and *Skadi Code Size - Genesys2* in *paper_artifact/README.md*.
Results: *paper_artifact/README.md* explains how to collect results.

(E3): [Load Times] [1 human-hour]: Determine the load time of Skadi compared to its baseline, Zephyr.
Preparation: Synthesize and program the FPGA configuration. Setup the software toolchain.
Execution: Follow the steps under *7.3: Load times—Load times zephyr - Genesys 2* and *Load times Skadi - Genesys 2* in *paper_artifact/README.md*. *Note:* We loaded each sample 5 times to determine a standard deviation, which ended up being negligible. You can reduce the number of repetitions in order to save time if you are confident in the significance of our result.
Results: The samples will print the load time onto the UART console.

(E4): [Micro Benchmarks] [20 human-minutes]: Determine the performance of Northcape/Bredi operations.
Preparation: Synthesize and program the FPGA configuration. Setup the software toolchain.
Execution: Follow the steps under *7.4: Microbenchmarks—Microbenchmarks - Genesys 2* in *paper_artifact/README.md*.
Results: The samples will report minimum, maximum, mean and standard deviation for each operation. CPU cycles are given in lines like this: *create microbenchmarks (Samples) discarded/min/max/avg/stddev ns: 0/622/635/629.273684/4.387643*, where *create* is the name of the operation. Operations module cycles are given in lines like this: *create microbenchmarks (ops only) (Samples) discarded/min/max/avg/stddev ns: 0/0/0/0.000000/0.000000*. They are only significant in the second sample built by the documented steps (the sample that includes *ops_cycles.conf*).

(E5): [Compute Benchmarks] [10 human-minutes + 1 compute-hour]: Determine the performance of Skadi and Bredi for compute workloads.
Preparation: Synthesize and program the FPGA configuration. Setup the software toolchain.
Execution: Follow the steps under *7.5 System: Compute Performance—zephyr Compute Performance - Genesys2*, *Skadi Compute Performance - Genesys2* and *Linux compute Performance - Both* in *paper_artifact/README.md*. *Note:* Reprogram the FPGA in order to allow it to boot into Linux.
Results: The stream benchmark will print a table of results. Coremark will print an individual score. For both benchmarks, our main paper reports the average score only.

(E6): [Network Benchmarks] [2 human-hours]: Determine the performance of Skadi and Bredi for network workloads.
Preparation: Synthesize and program the FPGA configuration. Setup the software toolchain.
Execution: Follow the steps under *7.6 System: Network Performance—zephyr Network Performance - Genesys2*, *Skadi Network Performance - Genesys2* and *Linux Network Performance - Both* in *paper_artifact/README.md*. *Note:* Reprogram the FPGA

in order to allow it to boot into Linux.

Results: The ping and iperf utilities print their results to stdout. The RX/TX duration and MQTT-TLS benchmarks print summaries similar to the micro benchmarks.

(E7): [IRQ Benchmarks] [20 human-minutes]: Determine the IRQ latency for Skadi and Zephyr.

Preparation: Synthesize and program the FPGA configuration. Setup the software toolchain.

Execution: Follow the steps under *7.7 System: Real-Time Capability—zephyr Real-Time Capability - Genesys2* and *Skadi Real-Time Capability - Genesys2* in *paper_artifact/README.md*. *Note*: Our original submission contained additional measurements, comprising a Linux equivalent of the benchmarks and a scheduler microbenchmark (latency_sample). These benchmarks were cut from the paper and need not be repeated.

Results: The IRQ benchmarks print summaries similar to the micro benchmarks.

A.5 Notes on Reusability

We intend Skadi and Bredi to serve as a platform for future computer-architectural research. To this end, we provide additional documentation for toolchain and physical setup and wiring of the boards in *README.md*. As mentioned previously, a slightly restricted version of the SoC can be evaluated on the cheaper Arty A7 board without requiring licenses for Vivado or AMD/Xilinx IP. All experiments mentioned above can be attempted on the Arty A7 as well.

We discuss porting software to Skadi in our artifact and provide code samples. To this end, please refer to software/include/northcape-zephyr-stack/doc/hardware/porting.

You can use Zephyr's menuconfig to review supported configuraton options for Skadi, depending on your application needs.

Finally, the Bredi-specific hardware components exist in an individual directory with a README and an UVM-based testbench. Please see hardware/include/northcape. Certain parameters of Bredi components can also be tuned in the block design of the Vivado GUI, e.g., cache algorithms and sizes.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Cross-National Information Attacks: A Two-Decade Analysis of Troll Behavior in Korea

Jaehong Kim[1,2,*] Hyeonseung Kim[1,*] Jiseon Kim[1,2]
Alice Oh[1] Thorsten Holz[2] Wonjae Lee[1,†] Meeyoung Cha[2,1,†]
[1]*Korea Advanced Institute of Science and Technology (KAIST), Daejeon, South Korea*
[2]*Max Planck Institute for Security and Privacy (MPI-SP), Bochum, Germany*

A Artifact Appendix

A.1 Abstract

This artifact contains the dataset, model checkpoint, and source code for the paper. The dataset consists of 112 million South Korean Naver News comments spanning 2006 to 2025, annotated with troll categories. The model checkpoint is a KcELECTRA-based hierarchical classifier distilled from GPT-4.1 annotations, with 18 output heads covering comment-level classification and span-level rationale extraction. The code provides a script for running inference with the trained model on Korean-language comment dataset, and the Jupyter notebook for visualizing the strategy analysis results.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The dataset contains news comment text that may include biased, offensive, or politically sensitive content authored by suspected influence operation accounts. Usernames are partially masked in the released dataset. No destructive steps or elevated privileges are required to run the artifact.

A.2.2 How to access

The artifact is available on Zenodo at `https://doi.org/10.5281/zenodo.20257085`. It contains `data.zip` (the full 112M-comment dataset), `model.pth` (the trained KcELECTRA checkpoint), and the source code.

A.2.3 Hardware dependencies

Inference requires a CUDA-enabled GPU. The reference environment uses an NVIDIA GeForce RTX 3090 24 GB with CUDA 12.5 on Ubuntu 20.04.6 LTS. At least 32 GB of system RAM is recommended for large-scale inference. The included 1,000-comment sample completes in under two minutes on a single GPU. Visualization (A.4.2) requires no GPU and runs on a standard CPU-based machine.

A.2.4 Software dependencies

The reference environment runs Ubuntu 20.04.6 LTS. Miniconda 26.1.1 is used to manage the environment; installation instructions are available at `https://docs.anaconda.com/miniconda/`. Python 3.9 or above is required. All dependencies are listed in `environment.yml`. The primary packages are PyTorch 2.6.0 (CUDA 12.4), Transformers 4.57.1, and Polars 1.26.0. The HuggingFace backbone `beomi/kcelectra-base` is downloaded automatically on first run. `environment.yml` specifies PyTorch with CUDA 12.4. Run `nvidia-smi` to check your CUDA driver version; if your CUDA driver version differs, update the `cu124` entries in `environment.yml` before running `conda env create` (e.g., `cu118` for CUDA 11.8).

A.2.5 Benchmarks

The full dataset (`data.zip`) is part of the artifact. A 1,000-comment username-masked sample (`data/sample.parquet`) is included in the code directory to support immediate functionality testing.

A.3 Set-up

A.3.1 Installation

Download and unzip `code.zip` from Zenodo and navigate to the extracted `code/` directory. `model.pth` and source code are in the root of that directory; `sample.parquet` is in the `data/` subdirectory. Create a Conda environment and install dependencies:

```
conda env create -f environment.yml
conda activate naver
```

A.3.2 Basic Test

Run inference on the included sample:

```
python inference.py
```

The expected output ends with:

```
Using device: cuda:0 Loaded 1000 samples from
data/sample.parquet
Saved 1000 predictions to inference/
predictions.json
```

A.4 Evaluation workflow

A.4.1 Major Claims

We designate the findings in Section 5 (*Strategy Analysis*) as the paper's main claims because they can be reproduced without access to identifying user information. By contrast, reproducing the reported model performance would require the original usernames, which cannot be released because doing so could identify individual users.

(C1): *Condemning Korea* is the dominant high-visibility rhetorical strategy among suspected troll accounts, as supported by three findings reported in Section 5 and verified by experiment (E1):

- *Condemning Korea* achieves higher like ratios than all other strategies (mean = 0.514), being the only strategy to exceed the neutral threshold, while *Condemning Rival* (0.422), *Praising MNS* (0.362), and *Praising Partner* (0.241) all fall below 0.5 (Figure 7a).
- As *Condemning Korea* rhetoric intensity increases, predicted like ratio rises above the neutral threshold at intensity = 0.45 ($\beta = 0.1086$, $p < 0.001$), whereas no other strategy significantly boosts like ratio (Figure 7b).
- Among high-visibility *Condemning Korea* comments (like ratio = 1), 7 of the top-10 most targeted entities are political leaders spanning both ideological camps, suggesting that targeting high-salience political figures amplifies engagement (Figure 8).

A.4.2 Experiments

(E1): *[Strategy Analysis]* [20–30 human-minutes + 10 compute-minutes + 3 GB disk + 32 GB RAM]: The aim of this experiment is to generate the figures present in the paper.

Preparation: Complete the installation steps in Section A.3.1. The required data files (`data/suspected_troll_df.parquet` and `data/suspected_troll_spans.parquet`) are included in `code.zip` and will be present in the `data/` subdirectory after extraction. Open `visualization.ipynb` and set the kernel to `naver`.

Execution: Run all cells in `visualization.ipynb` using the button "Run All Cells".

Results: Resulting figures should match Figures 7(a), 7(b), and 8 present 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/usenixsec2026/.

ARTIFACT EVALUATED
USENIX
AVAILABLE

USENIX Security '26 Artifact Appendix: Characterizing Security and Privacy Teaching Standards for Schools in the United States

Katherine Limes
Colorado School of Mines

Nathan Malkin
New Jersey Institute of Technology

Kelsey R. Fulton
Colorado School of Mines

A Artifact Appendix

A.1 Abstract

The artifact contains the dataset, analysis code, and qualitative codebook behind the claims in our paper "Characterizing Security and Privacy Teaching Standards for Schools in the United States". The dataset is a SQLite database containing all standards, with a table for all 37 state standards and eight national organization standards analyzed. The dataset includes our annotations, such as the qualitative labels assigned during analysis. The analysis code is a Jupyter Notebook which runs all database queries and analysis on their outputs used for the study. The codebook includes definitions, category descriptions, and examples for the 109 S&P topics identified in the study.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There is no risk to evaluators from downloading our artifacts or executing our code.

A.2.2 How to access

Follow the link to the OSF archive here: https://osf.io/hwkuc/overview?view_only=c226130f34fb4c28be314426390d6656

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

A Linux, macOS, or Windows machine with uv (https://docs.astral.sh/uv/) installed. uv provisions the required Python interpreter (3.13) and all notebook dependencies (Jupyter, nbconvert, ipykernel) from the pinned uv.lock; no other software needs to be installed manually. The analysis logic itself uses only the Python standard library (sqlite3, csv).

A.2.5 Benchmarks

None.

A.3 Set-up

Install uv (https://docs.astral.sh/uv/getting-started/installation/) if needed.

A.3.1 Installation

Download SQL Queries.ipynb and standards_updated1 to the same directory, from the provided link.

A.3.2 Basic Test

Open SQL Queries.ipynb using uv (uv run jupyter notebook "SQL Queries.ipynb"). Run the first two cells, either with the run button or Ctrl+Enter, the second cell should output:

Troubleshooting 39
Networking 38
Networking Security 35
Encryption 34
Safe & Appropriate Behavior 41
Tracking and Digital Footprint 39
Content and Information 25
Cloud Computing 13
Embedded Systems 23
Organizational Security 25
Attacks & Attackers 37
Code Debugging 36
Secure Coding 31
Defenses 41
Laws and Intellectual Property 43
Equity, Accessibility, and Diversity 44
Real World Impacts 37
Other 22

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *We labeled 12,201 standards from all U.S. states and eight national organizations, finding that 3,920 of them are related to security and privacy.*

(C2): *After qualitative analysis, we identified 109 different security and privacy related topics in the standards, and counted all of the states and national organizations in which they appear. See Table 1 in the paper.*

(C3): *States and national organizations categorized 1,669 of the 3,920 standards that we identified as security and privacy related as security and privacy related. They labeled privacy-related standards significantly less often than security-related standards.*

(C4): *Even popular topics are not universally covered by states and national organizations. Coverage varies significantly by grade level.*

A.4.2 Experiments

(E1): *[5 human-minutes + 5 compute minutes + 5MB disk]:*
How to: *Run all cells in SQL Queries.ipynb*
Preparation: *Install uv, if needed.*
Execution: *Run all cells in SQL Queries.ipynb*
Results: [C1] will be printed by the fifth cell, the output should be *Security Related: 3920*
Total: 12201, which correspond to the number of standards classified as security related and the total number of standards analyzed, respectively. These numbers appear in Section 4 of the paper, as well as the abstract.
Results of [C2] will be written to code_counts.tsv by the second cell. Values in code_counts.tsv should match those entered in Code Definitions + Example Standards.xlsx, and correspond to Table 1 in the paper. Counts for categories, rather than individual codes (i.e., *Encryption* vs *Broad coverage of encryption*) will be printed by the first cell, as described in Section A.3.2 above.
[C3] will be printed by the final cell. The printed output should begin with *Total Number of Standards Categorized as S&P: 1669*, and then a list of topic labels and decimal values between 0 and 1, which represent the percentage of each topic name classified as security in states which classified any topic as security. These correspond to the claims in Section 4.2 in the paper.
[C4] uses cells three, four, and six. Cell three prints a category of topic, followed by the states and organizations which do not have any topics from that category in their standards. Cell four prints the number of S&P standards found at each grade band, reported in Section 4 of the paper, as well as the counts of each topic at a grade band, used for the findings in Section 4.3.2. Cell six prints the name of each state or national organization, followed by how many of the fourteen most common topics that were found in that state or organization's standards, used for the analysis in Section 4.3.1 in the paper.

A.5 Notes on Reusability

All standards and labels are contained in standards_updated1.db, with each state having its own table, and each standard having a row in that table. Each table has eleven columns: name (the state's identifier for the standard), grade_level (grade level or band the standard is intended to be taught at), category (category assigned to the standard by the state), category_detail (any explanation for the category provided by the state), standard (text of the standard), standard_detail (additional detail, clarification, or examples attached to the standard by the state), code (qualitative label added by the research team), security_category (is the category assigned by the state related to security or privacy, Y/N?), security_mention (does the standard mention security or privacy, Y/N?), has_examples (does the standard include examples, Y/N?), and is_required (does the state require CS for graduation, Y/N?). This database can be queried freely for future experiments.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: BugAuditor: Detecting Bugs via Inconsistent Defensive Code Auditing

Miaoqian Lin
The University of Hong Kong

Kai Chen
Institute of Information Engineering, Chinese Academy of Sciences

Hao Chen
The University of Hong Kong

A Artifact Appendix

A.1 Abstract

BugAuditor is an LLM-driven bug detection framework that uses inconsistent defensive handling as a new oracle for detecting project-specific bugs. Its key insight is that large software systems already contain abundant defensive code, where developers apply defensive operations to prevent bugs in security-sensitive contexts. When similar security-sensitive behaviors are handled defensively in some places but not in others, the inconsistency may indicate a real bug.

BugAuditor first identifies defensive code snippets across the codebase, then infers defensive patterns that capture both the security-sensitive behavior and the required defensive handling. It finally applies these patterns to audit similar code contexts and detect inconsistent defensive handling.

This artifact includes the implementation of BugAuditor, key experimental data , and usage documentation. It aims to demonstrate that BugAuditor can effectively locate defensive code in large-scale systems, mine defensive patterns, and detect new bugs.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact poses no risks to the evaluators' machine in terms of security, data privacy, or any other ethical concerns

A.2.2 How to access

You can access the artifacts via the following stable link to Zenodo: https://doi.org/10.5281/zenodo.20267685 You can also access it in Github: https://github.com/Yuuoniy/BugAuditor The artifact is licensed under Apache License 2.0.

A.2.3 Hardware dependencies

At least 4 CPU cores recommended (20+ better). At least 16 GB RAM recommended. At least 15 GB free disk space recommended. BugAuditor works on standard machines. It is implemented in a multithreaded manner, and we recommend using a multithreaded computer to speed up the evaluation process. In our evaluation, we conducted experiments on a 64-bit Ubuntu server equipped with 212 GB of memory, 2 TB of storage, and an Intel Xeon Gold 5218 CPU with 64 cores.

A.2.4 Software dependencies

BugAuditor is implemented in Python and leverages existing code analysis tools, including Joern, Tree-sitter, and Weggli. We provide a Dockerfile to automate the setup and creation of the Docker environment. This Dockerfile includes everything needed to configure the runtime environment, install third-party tools and dependencies, set up Python libraries.

The artifact requires access to LLM APIs. The overall token cost is very low and is convenient for evaluation. The paper uses DeepSeek-V3.2 deployed locally. Due to the official DeepSeek API no longer providing access to DeepSeek-V3.2, we recommended DeepSeek-V4-Flash (via api.deepseek.com) as a convenient replacement. This model produces similar results while maintaining equal or lower cost. DeepSeek-V3.2 can still be used for testing via other available OpenAI-compatible endpoints.

A.2.5 Benchmarks

Source code of the Linux kernel.

A.3 Set-up

A.3.1 Installation

We provide a `Dockerfile` to simplify the installation process. Users can build the BugAuditor Docker image using the provided `Dockerfile` and download the necessary programs for testing. Detailed instructions are available in the `INSTALL.md` file. Please refer to it and follow the steps accordingly.

A.3.2 Basic Test

We provide a minimal running example for a quick sanity check of BugAuditor. This example uses `clk_put` as the seed defensive operation, reproducing the working flow of

defensive code locating, defensive pattern reasoning, and bug auditing on a small input.

1. Basic test for defensive pattern reasoning. [2 human-minutes + 5 compute- minutes]: Run the following command:

```
$ bash artifact/minimal/run_pattern_reasoning.sh
```

The script locates defensive-code snippets for `clk_put`, selects 10 examples, and runs defensive pattern reasoning. The expected terminal output is provided in the file `expected_pattern_reasoning_output.txt` located at `artifact/minimal/reference/`. The main generated results are stored under `artifact/results/minimal/`:

- `defensive_code_snippets.json`
- `inferred_defensive_patterns.json`

The inferred patterns cover multiple security-sensitive operations, including `of_clk_get_by_name`, `clk_hw_get_clk`, `of_clk_get`, `clk_get`, `of_clk_get_from_provider`, and `clk_get_sys`. Note that the minimal example is not meant to exactly reproduce Table 9 in the paper.

2. Basic test for bug auditing. [2 human-minutes + 5 compute- minutes]: Run the following command:

```
$ bash artifact/minimal/run_bug_auditing.sh
```

This test uses one mined defensive pattern, `of_clk_get_by_name -> clk_put`, to audit a focused set of `clk_put`-related cases. It detects buggy functions such as `lpc32xx_clk_init` and `nmdk_timer_of_init`. The key results are written to the `artifact/results/minimal/` directory, including the generated bug report file `bug_reports_clk_put.json`.

A.4 Evaluation workflow

A.4.1 Major Claims

The main workflow uses a given defensive operation as a seed to locate similar defensive code snippets across the codebase (e.g., Linux kernel). It then reasons about the defensive patterns and applies them to detect new bugs. Overall, the process consists of two phases: defensive code locating and pattern reasoning, followed by bug detection.

(C1): BugAuditor can locate a large number of defensive code snippets and infer the corresponding defensive patterns, using six given defensive operations as seeds. This is supported by the experiments (E1).

(C2): Using the inferred defensive patterns, BugAuditor detects dozens of bugs in the Linux kernel. This is validated by the experiments (E2).

A.4.2 Experiments

The experiments were conducted on a reduced scale for faster and more convenient evaluation. Complete experiment scripts and detailed instructions are also provided in `AE.md`.

(E1): [Defensive Pattern Reasoning] [5 human-minutes + 1 compute-hour]:
This experiment reproduces the defensive pattern reasoning stage on the six seed defensive operations used in the paper. It involves defensive code collection and the corresponding pattern reasoning.
Preparation: Follow the previously instructions for installation and downloads the source code of Linux kernel. Run the basic test to check the environment. Make sure setup the LLM endpoint.
Execution: Please run:
`$./artifact/r_pattern_reasoning/run.sh`
Results: The key results are saved under `artifact/results/r_pattern_reasoning/`. `inferred_patterns.csv` lists all defensive patterns inferred, with one pattern per row.

(E2): [Bug Auditing] [5 human-minutes + 30 compute-minutes]:
Preparation: Follow the artifact installation instructions and run the minimal example first to check the environment. The LLM endpoint should be configured before running this stage.
Execution: Please run:
`$ ./artifact/r_bug_auditing/run.sh`
Results: The key results are saved under `artifact/results/r_bug_auditing/`, including the main output file `bug_reports.json`.

A.5 Notes on Reusability

To facilitate artifact evaluation, we designed the reduced-scale evaluations that reduces time and token consumption. We additionally provide instructions for applying BugAuditor to other programs, such as OpenSSL and FFmpeg. Please refer to `AE.md` for details.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: FIRMREBUGGER: A Benchmark Framework for Monolithic Firmware Fuzzers.

Mathew Duong[1], Michael Chesser[1], Guy Farrelly[1], Surya Nepal[2], and Damith C. Ranasinghe[1]

[1]*University of Adelaide*
[2]*Data61 CSIRO*

A Artifact Appendix

A.1 Abstract

FIRMREBUGGER is a bug-oracle–based evaluation framework and benchmark suite for monolithic firmware fuzzers that enables automated, fair, and reproducible assessment of bug discovery. It supports hard-to-measure metrics such as time-to-bug, which are computed, visualized, and reported using C-based bug oracles included with the artifact. The fuzzers evaluated in the paper are provided with Dockerfiles and automated build scripts to facilitate reproduction. Additionally, the artifact includes a web-based interface for scheduling and managing fuzzing runs and for visualizing the reported metrics.

A.2 Description & Requirements

The full evaluation of all nine state-of-the-art monolithic firmware fuzzers reported in the paper requires approximately 10 CPU-years. Due to this substantial computational cost, the number of trials, fuzzing duration, and selection of fuzzers and binaries can be adjusted as needed during artifact evaluation. Hardware requirements vary by fuzzer, but in general each fuzzing instance requires approximately 2 GB of RAM. For CPU resources, any modern x86 processor is sufficient, with one fuzzer instance typically assigned per physical CPU core.

A.2.1 Security, privacy, and ethical concerns

During setup, network access is required to fetch dependencies from external sources (e.g., GitHub). Docker is used to isolate and execute individual fuzzer runs and therefore requires a functional Docker environment. All previously unknown bugs have been responsibly disclosed. Any additional bugs discovered during artifact evaluation should be handled following responsible disclosure guidelines.

A.2.2 How to access

All code and data are publicly available at:

- **FirmReBugger stable release:** https://figshare.com/s/d37c9a1cb67b6d07be0c

We also have a GitHub repository, with the version corresponding to the USENIX submission provided as a tagged release:

- **FirmReBugger GitHub repository:** https://github.com/FirmReBugger/FirmReBugger/releases/tag/USENIX-Artifact-Evaluation

A.2.3 Hardware dependencies

The experiments do not require specialized hardware and can be run on a generic x86-64 system with adequate RAM and storage. The results reported in the paper were obtained on an AMD Ryzen Threadripper 5995WX system with 256 GB of RAM running Ubuntu Server 22.04.

A.2.4 Software dependencies

Our experiments require a Linux-based operating system with the following software dependencies: Bash, Docker, C compiler, Python, npm and UV.

The evaluation also depends on the following fuzzing frameworks and tools:

- Ember-IO-Fuzzing
- Fuzzware
- Fuzzware-Icicle
- SEmu
- DICE
- Hoedur
- GDMA
- SplITS
- Hoedur

A.2.5 Benchmarks

The artifact includes the FIRMBENCH benchmark, consisting of 61 real-world monolithic firmware binaries and 280 documented bugs, as well as C-based bug oracles. This data is required to reproduce the experiments reported in the paper.

A.3 Set-up

The setup has been tested on a clean installation of `Ubuntu Server 22.04`. Certain fuzzers require adjustments to system parameters. Specifically, enable core dumps and increase the limits for inotify watches and instances, which are necessary for the operation of some fuzzers included in this artifact. Note that `Docker` may require `sudo` privileges, unless your user has been added to the `docker` group. For more details, refer to the README at https://github.com/FirmReBugger/FirmReBugger.

A.3.1 Installation

After ensuring the initial setup, the following software must be installed:

- `uv` (https://docs.astral.sh/uv/getting-started/installation/)
- `docker` (https://docs.docker.com/engine/install/)
- `npm` (version $\geq$ 22.0.0; see https://docs.npmjs.com/downloading-and-installing-node-js-and-npm)

In addition, please install the following packages:

```
Install uv https://docs.astral.sh/uv/
    getting-started/installation/
Install docker https://docs.docker.com/
    engine/install
sudo apt-get install automake texinfo
    unzip
sudo apt install gcc-arm-none-eabi
sudo apt install texlive-latex-base
sudo apt install texlive-xetex texlive-
    fonts-recommended texlive-latex-
    extra
install npm (npm >= 22.0.0)
```

And set up the host for fuzzing: Set up the host for fuzzing:

```
echo core | sudo tee /proc/sys/kernel/
    core_pattern
sudo sh -c echo 524288 > /proc/sys/fs/
    inotify/max_user_watches && echo
    8192 > /proc/sys/fs/inotify/
    max_user_instances
```

A.3.2 Build All Fuzzers

If you're having trouble building the fuzzers locally or do not need to, you can pull our prebuilt images from GHCR:

```
uv run frb build --use-prebuilt
```

This pulls registry images and tags them locally as `frb:<fuzzer>` and `frb_original:<fuzzer>`, which are the names used by fuzzing/triage runtime.

Fall back to build locally If the prebuilt images fail to run on your system, fall back to the non-prebuilt images with:

```
cd FirmReBugger
export FIRMREBUGGER_BASE_DIR=$(pwd)
uv run frb build
```

A.3.3 Web Application (Recommended Approach)

Launch the web application with the following commands:

```
uv run frb app --help
uv run frb app -p <port>
```

- **Report:** Provides a summary of your fuzzing campaigns using FirmReBugger.
- **Job Manager:** Allows you to configure and schedule fuzzing or triaging jobs automatically.

A.3.4 Available Commands

The following commands are available for reference:

```
cd FirmReBugger
export FIRMREBUGGER_BASE_DIR=$(pwd)
uv run frb --help
uv run frb fuzz --help
uv run frb build --help
uv run frb bug-analyzer --help
uv run frb charting-tool --help
uv run frb app --help
```

A.3.5 Basic Test

To verify your installation, first build a few fuzzers using the prebuilt images:

```
uv run frb build --use-prebuilt
```

Next, start the web application (served on `localhost:5000`; if you are connected remotely, you may need to forward the port):

```
uv run frb app
```

The web interface provides a summary report of your active and completed fuzzing campaigns using FirmReBugger, including interactive visualizations and management tools.

The *Job Manager* allows you to configure and schedule fuzzing or triaging jobs.

Recommended usage: Launch fuzzing and triaging directly through the web app interface. This ensures that inputs, job scheduling, and result visualization are handled automatically.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *A comprehensive evaluation of 7 state-of-the-art monolithic firmware fuzzers (Ember-IO, Fuzzware, Fuzzware-Icicle, SEmu-Fuzz, SplITS, Hoedur, and MultiFuzz) was conducted on the FirmBench dataset comprising 31 binaries. The results of this evaluation can be found at our GitHub repository[1], and underlying data is used in Figures 9, and 12 of the paper.*

(C2): *We benchmarked 6 state-of-the-art monolithic firmware fuzzers (Ember-IO, Fuzzware, Fuzzware-Icicle, SplITS, Hoedur, and MultiFuzz) on the FirmBenchX dataset, comprising 22 additional binaries. These results can be accessed at our GitHub repository[2], and the corresponding analysis is presented in Figures 10, and 12.*

(C3): *We evaluated two state-of-the-art DMA-centric firmware fuzzers (GDMA and DICE) on the FirmBenchDMA dataset, which includes two binaries. These results are summarized in Table 1 and contribute to the analyses found in Figure 10.*

A.4.2 Experiments

(E1): **How to:** Open the FirmReBugger web app and use the Job Manager to schedule fuzzing jobs for the FirmBench suite using each of the 7 fuzzers. Configure each job for 10 trials per binary, each running for 24 hours. Once fuzzing is complete, queue triaging jobs for each fuzzer via the "More Actions" button in the completed section. Visualize the results using the Report page.
Preparation: Ensure that all dependencies and environment configurations described in Section A.3.1 have been completed.
Execution: In the FirmReBugger base directory, launch the web app with:

```
uv run frb app
```

Open the web browser at `localhost:5000`, navigate to "Job Manager," and schedule fuzzing jobs for all FirmBench binaries, assigning each of the listed fuzzers.
Results: After both fuzzing and triaging are completed, review results on the Results/Report page. The outputs should qualitatively align with those reported in the GitHub repository and match summary data in Figures 9, 10, and 12 of the paper.

(E2): **How to:** Use the Job Manager in the FirmReBugger web app to schedule fuzzing and triaging for the FirmBenchX benchmark (22 binaries) using Ember-IO, Fuzzware, Fuzzware-Icicle, SplITS, Hoedur, and MultiFuzz, with 10 trials per binary for 24 hours each. Visualize all results as in E1.
Preparation: As above, confirm all dependencies and environmental setup from Section A.3.1.
Execution: Launch FirmReBugger, open the web interface, and repeat the scheduling and triage process for the FirmBenchX suite and listed fuzzers.
Results: Review results in the Report page, and compare with published results on GitHub and in Figures 9, 10, and 12.

(E3): **How to:** In FirmReBugger's Job Manager, schedule fuzzing and triage tasks for GDMA and DICE against the two FirmBenchDMA binaries, with recommended durations and trial counts. Inspect and visualize the results as with the other benchmarks.
Preparation: Complete all installation and environment setup as per Section A.3.1.
Execution: Launch the web app, open the Job Manager, and schedule the relevant jobs for the DMA-centric evaluation.
Results: Result summaries will be available through the Reports page; validate your findings against Table 1 in the paper.

A.5 Notes on Reusability

FirmBench is designed to be a living and extensible benchmarking platform for firmware fuzzing. We encourage and support continued community contributions, enabling the benchmark suite to grow and remain relevant as new fuzzing techniques and bug types emerge.

The artifact provides mechanisms for users to submit new bugs ("Ravens") and new binaries, together with a thorough root cause analysis, for potential inclusion in FirmBench. Community review and validation ensure that each Raven accurately encapsulates the bug semantics before being merged. This process maintains the integrity and utility of the benchmark for future research.

As the field evolves, FirmBench allows for the inclusion of focused subsets, such as new categories of bugs or benchmarks guarded by specific "roadblocks" (e.g., DMA-centric bugs), to support evaluation of targeted fuzzing techniques. The infrastructure is flexible and can accommodate the creation of balanced binary sets with diverse bug types and challenge properties.

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/usenixsec2026/.

[1] https://github.com/FirmReBugger/FirmReBugger/blob/main/paper_results/FirmBench.pdf

[2] https://github.com/FirmReBugger/FirmReBugger/blob/main/paper_results/FirmBenchX.pdf





USENIX Security '26 Artifact Appendix: RecStruct: Recovering Nested Struct Types from Stripped Binaries via Stack-Driven Unification

Yuxin Chen, Zhiyang Fang, Shiyi Wu, Yixin Xu, Yuhang Wang, Xiaokang Yin, Junfeng Wang

A Artifact Appendix

A.1 Abstract

The artifact contains the complete implementation of RecStruct, a tool for recovering nested and recursive structure types from stripped binaries using Stack-Driven Unification. It includes the Rust source code, seven benchmark datasets with stripped inputs and pre-extracted YAML ground truth, and tools for running the analysis and computing the metrics used in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. The experiments analyze pre-compiled open-source binaries and do not process sensitive data or modify system configuration. Network access is only needed during setup if Cargo dependencies are not already cached.

A.2.2 How to access

The artifact is archived on Zenodo: version DOI https://doi.org/10.5281/zenodo.20628819; concept DOI https://doi.org/10.5281/zenodo.20255915.

A.2.3 Hardware dependencies

The reference experiments used Ubuntu 24.04 on a server with 32 vCPUs and 46GB RAM. We recommend 16GB RAM for the small and medium datasets, 32GB RAM for QEMU and OpenSSL, and at least 20GB free disk space for the downloaded archives, extracted datasets, builds, and results. No GPU or specialized hardware is required.

A.2.4 Software dependencies

- Linux x86-64 (Ubuntu 22.04 or later; tested on Ubuntu 24.04)
- Rust toolchain $\geq$ 1.88 (edition 2024; tested with Rust 1.93)
- Cargo and standard build tools (e.g., Ubuntu's `build-essential`)
- Python $\geq$ 3.10 only for optional auxiliary scripts

Cargo fetches and builds all Rust dependencies. IDA is not required for the experiments below because the archive includes pre-extracted ground truth; the IDA script is provided only for optional ground-truth re-extraction.

A.2.5 Benchmarks

The archive contains `coreutils-O1`, `coreutils-O2`, `coreutils-O3`, `utils`, `binutils`, `qemu`, and `openssl`. Each dataset contains `bin/` (with debug information), `strip/` (the stripped analysis inputs), and `groundtruth.yaml`.

A.3 Set-up

Download `RecStruct-source.tar.gz` and `RecStruct-dataset.tar.gz` from Zenodo, place them in an empty directory, and run:

```
tar -xzf RecStruct-source.tar.gz
tar -xzf RecStruct-dataset.tar.gz
cd RecStruct/struct-recover-rs
cargo build --release
cd ../struct-recover-eval/tools
cargo build --release
```

The commands below assume the resulting layout in which `RecStruct/` and the seven dataset directories share a parent directory. They explicitly set `-stride` and `-output`; this is required because the legacy defaults in version 5 contain `/home/user` paths.

A.3.1 Basic Test

From `RecStruct/struct-recover-rs`, run:

```
./target/release/stride pipeline \
  --bin testcase/old-case --mode full \
  --output target/release/output
```

The command should finish without error after analyzing 24 functions and write `old-case.h` and `old-case.yaml` under `target/release/output/`. The output contains the four principal interdependent structures and their nested/cyclic pointer relationships shown in Figure 3 and Table 2 of the paper (additional internal type groups are also reported).

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): RecStruct executes end-to-end on a stripped ELF binary and recovers nested and recursive structure definitions (RQ1, Figure 3 and Table 2). This is exercised by E1.

(C2): RecStruct's dataset pipeline computes layout accuracy, nesting recognition, and Trex-style hierarchical scores (RQ2, Tables 3–5). This is exercised by E2.

(C3): RecStruct records end-to-end runtimes and supports analysis of large binaries with a two-hour timeout per binary (RQ3–RQ4, Tables 6–8). This is exercised by E2.

(C4): The runner exposes LOCAL, INTRA, and GLOBAL analysis scopes used by the Stack-Driven Unification ablation (RQ5, Figure 6). This is exercised by E3.

A.4.2 Experiments

(E1): End-to-end micro-benchmark [5 human-minutes + less than 1 compute-minute]. Run the Basic Test and inspect `old-case.h` and `old-case.yaml`. Successful file generation and the four mapped structures validate C1.

(E2): Dataset-scale analysis and evaluation [10 human-minutes + dataset-dependent compute time]. From `RecStruct/struct-recover-eval/tools`, run, for example:

```
D=coreutils-O1
V=baseline-ae
STRIDE=../../struct-recover-rs/target/\
release/stride
./target/release/run-dataset \
  --dataset ../../../$D \
  --variant "$V" \
  --stride "$STRIDE" \
  --output ../results
./target/release/run-eval \
  --dataset ../../../$D \
  --results ../results/$D-stride-$V
```

Replace `coreutils-O1` in both commands with another dataset name to evaluate it. The first command creates `all.yaml`, `summary.md`, and `run_stats.json`. The second prints an evaluation summary and creates `eval2`, `eval3`, and `eval4` reports in Markdown/JSON plus `eval_summary.json`. Eval2 corresponds to layout accuracy (Table 3), Eval3 to nesting recognition (Table 4), and Eval4 to the Trex-style score (Table 5). On the archived inputs, representative F1 scores are approximately 42%/17% for `coreutils-O1` and 18%/5% for `utils` (layout/nesting); minor differences may occur across toolchains. Runtime is machine-dependent: the reference totals are about one minute for `coreutils-O1`, 1.95 hours for `binutils`, and 19.9 aggregate compute-hours for `qemu`. One QEMU binary may reach the documented two-hour timeout.

(E3): Analysis-scope ablation [5 human-minutes + three E2 runs]. Repeat the first E2 command with `-analysis-mode 0`, `1`, and `2`, using distinct variant names such as `ablation-local`, `ablation-intra`, and `ablation-global`; then run `run-eval` on each resulting directory. These modes correspond to LOCAL, INTRA, and GLOBAL in Figure 6 and exercise C4. Detailed baseline and batch-analysis instructions are in the archived README.

The artifact received the Artifacts Available and Artifacts Functional badges; Results Reproduced was not applied for. The commands above therefore document the functional evaluation workflow and do not assert a Results Reproduced badge.

A.5 Version

Based on the Artifact Evaluation template V20231005. The evaluation and badging methodology is documented at https://secartifacts.github.io/usenixsec2026/.



USENIX Security '26 Artifact Appendix: CombiSan: Unifying Software Sanitizers for Comprehensive Fuzzing

Matteo Marini*
Sapienza University of Rome
m.marini@diag.uniroma1.it

Floris Gorter*
Vrije Universiteit Amsterdam
f.c.gorter@vu.nl

Daniele Cono D'Elia
Sapienza University of Rome
delia@diag.uniroma1.it

Cristiano Giuffrida
Vrije Universiteit Amsterdam
giuffrida@cs.vu.nl

* Equal contribution joint first authors

A Artifact Appendix

A.1 Abstract

This artifact provides the instructions to run and evaluate CombiSan, a fuzzing-optimized software sanitizer that combines detection of addressability issues, use-of-uninitialized-memory (UUM) errors, and other undefined behavior bugs. Our results show that CombiSan can do so with relatively low overhead, while matching the accuracy of the three most used software sanitizers (ASan, MSan, and UBSan).

A.2 Description & Requirements

The artifact consists of a modified LLVM 20.1.0 framework, which is used to instrument the targets to enable bug detection, a modified AFL++ 4.32c project that supports *deferred detection* of bugs, and infrastructure to test CombiSan. The performance tests require the SPEC CPU benchmarking suite, which is a licensed product we cannot redistribute.

A.2.1 Security, privacy, and ethical concerns

This artifact does not pose concerns to evaluators. Broader ethical considerations are discussed in the paper.

A.2.2 How to access

The source code of CombiSan, and the infrastructure to test it, is available at `https://github.com/vusec/combisan`, where it will be maintained. The GitHub `README.md` and `AE.md` pages contain more instructions. We provide permanent storage at `https://zenodo.org/records/16949365`.

A.2.3 Hardware dependencies

CombiSan supports commodity hardware. We recommend at least 16 GB of RAM to build LLVM. Additionally, we recommend at least 50 GB of disk space for the installations of LLVM and AFL++, as well as the Docker images.

A.2.4 Software dependencies

CombiSan can run on any modern installation of a Linux-based OS that is capable of compiling and running LLVM 20.1.0 and AFL++ 4.32c, such as any recent Ubuntu version ($\geq$20.04). We evaluated on Ubuntu 24.04 LTS. To support the compilation of its components, common development tools are required (e.g., `gcc`/`g++`, `cmake`, `ninja`); they can be easily found in every modern package manager. For example, using

```
sudo apt install gcc g++ cmake ninja-build \
unzip python3-terminaltables
```

To facilitate fuzzing, we used Docker; this artifact contains the Dockerfiles and accompanying build instructions.

A.2.5 Benchmarks

CombiSan was evaluated using three major benchmarks. First, performance was evaluated using the SPEC CPU benchmarks. Then, we assessed accuracy capabilities using the NIST Juliet Test Suite. Lastly, we tested fuzzing capabilities using a custom dataset of common fuzzing targets.

A.3 Set-up

All the information below is also discussed in the documentation. Please refer to `AE.md` for more detailed instructions.

A.3.1 Installation

To start using CombiSan, clone its repository with:

```
git clone --recursive \
https://github.com/vusec/combisan.git
```

Then build two versions of LLVM, CombiSan's and an unmodified one to enable comparisons, with:

```
cd combisan
./build_llvm.sh
./build_llvm_clean.sh
```

Finally, set environment variables `CLEAN_LLVM_DIR` and `SPEC_DIR` to the paths of the newly-compiled, clean version of LLVM and to the SPEC installation, respectively.

A.3.2 Basic Test

To use COMBISAN, use the `clang(++)` binary of the newly built LLVM project, and add the `-fsanitize=address` compilation flag on any C(++) program. Since COMBISAN's code lives alongside ASan's inside the LLVM codebase, activating ASan's compilation flag enables COMBISAN's detection of both addressability issues and UUM errors. To also enable detection of UB bugs, add the desired flags (e.g., `-fsanitize=undefined` to detect all of them).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): COMBISAN simultaneously detects addressability issues, UUM errors, and UB bugs. It does so while matching the detection capabilities of ASan, MSan, and UBSan. This is proven by experiment (E1).

(C2): COMBISAN incurs low overhead compared to running sanitizer disjointly, as shown in experiment (E2).

(C3): COMBISAN integrates its detection capabilities while fuzzing. In particular, COMBISAN uses *deferred detection* of bugs to prevent early bugs from masking later ones. This is demonstrated by experiment (E3).

A.4.2 Experiments

(E1): [1 compute-hour]: Bugs detection accuracy.

How to: Run the tests from the Juliet Test Suite with COMBISAN, ASan, MSan, and UBSan.

Preparation: Compile COMBISAN's LLVM version with default configuration. This can be done using the `build_llvm.sh` script. Note that this script removes any other compiled version of COMBISAN's LLVM.

Execution: Execute the following command:

```
python3 setup.py run juliet csan_O0 \
--build --parallel=proc \
--parallelmax=$(nproc) \
--cwe 121 122 124 126 127 415 416 457 190 \
191 194 476 758 843 &> juliet_out
```

Notice how these tests are performed with `O0` as higher optimization levels (e.g., `O2`) mask bugs in Juliet. Additional tests can be made by changing the target tool from `csan_O0` to `asan_O0`, `msan_O0`, and `ubsan_O0`.

Results: The results can be filtered out with:

```
cat juliet_out | grep Passed
```

This reproduces the experiment from Section 6.2, which also discusses the minor differences in bug count.

(E2): [6 compute-hours]: Slowdown and memory overhead.

How to: Run the SPEC CPU benchmarks with COMBISAN, ASan, MSan, and UBSan.

Preparation: Compile COMBISAN's LLVM version with the *recovery* configuration detailed in Section 6.3. This can be done using the `build_llvm_SPEC.sh` script. This script removes any other compiled version of COMBISAN's LLVM.

Execution: Execute the following command:

```
python3 setup.py run spec2017 baseline_O2 \
asan_O2 csan-rec_O2 msan-rec_O2 \
--build --parallel=proc --parallelmax=1
```

Change `spec2017` to `spec2006` to change version.

Results: The results can be obtained with:

```
python3 ./setup.py report spec2017 \
results/last/ --aggregate geomean \
--field runtime:median maxrss:median
```

This reproduces the experiment from Section 6.3.

(E3): [1 compute-hour]: Fuzzing and deferred detection.

How to: Perform fuzzing with COMBISAN, and leverage deferred detection to handle multiple bug classes.

Preparation: Build COMBISAN's Docker image with:

```
cd docker/combisan/
docker build -t combisan .
```

Note that this needs to build COMBISAN's LLVM, so it may take some time. Then build a FTS subject's image with, e.g.:

```
cd docker/fts/libxml2/libxml2_combisan/
docker build -t libxml2_combisan .
```

Execution: Spawn a container and fuzz with:

```
docker run -it libxml2_combisan bash
$AFL/afl-fuzz -i in -o out -m none -- \
$EXECUTABLE_NAME_BASE
```

Note how the command to start fuzzing is generic and can be used on every target image we provide. For three projects (`libxml2`, `pcre2`, and `re2`), an additional seed directory (`crashes/`) is available to test fuzzing in the presence of bugs, showing deferred detection in action. The bugs present in this directory are two duplicates of a bug from each sanitizer (ASan, MSan, and UBSan).

Results: COMBISAN can successfully support fuzzing while detecting multiple classes of bugs. Further, deferred detection enables deeper fuzzing in case bugs of certain classes are detected, as demonstrated when fuzzing is started with duplicated crashing inputs: addressability issues always cause a crash, UUM errors trigger a slow path check on novelty, and UB bugs only result in a crash the first time they are encountered.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: InstantOMR: Oblivious Message Retrieval with Low Latency and Optimal Parallelizability

Haofei Liang
Shanghai Jiao Tong University

Zeyu Liu
Yale University

Eran Tromer
Boston University

Xiang Xie
Primus Labs

Yu Yu
Shanghai Jiao Tong University

A Artifact Appendix

A.1 Abstract

This artifact contains our end-to-end code for our Oblivious Message Retrieval (OMR) construction, InstantOMR. InstantOMR is a construction based on the TFHE homomorphic encryption system. Our artifact demonstrates the performance of InstantOMR when implemented with the Primus-fhe library. Below, we discuss how to use our code and our main claims about its efficiency.

A.2 Description & Requirements

This section lists all the information necessary to recreate the same experimental setup we have used to run your artifact.

A.2.1 Security, privacy, and ethical concerns

We do not see any direct security, privacy, or ethical concerns. Specifically, our code can be run independently and does not need to interact with any other software or environment outside.

A.2.2 How to access

The artifact is available at Zenodo: https://doi.org/10.5281/zenodo.17959835.

A.2.3 Hardware dependencies

Our code can be compiled and run without specific hardware. However, one thing to note is that we benchmark InstantOMR scheme using Google Compute Cloud "c3d-highcpu-360", equipped with 180 cores and AVX-512 instructions. The numbers reported in our main claims are obtained using this instance. Other instances can also be used, but may result in a different performance.

A.2.4 Software dependencies

We have tested our code on Windows (MSVC toolchain) and Ubuntu (20.04/22.04).

To compile and run our code, the first step is to install build tools. On Windows, please install Visual Studio C++ Build tools. On Ubuntu and Debian, please install build-essential according to the instructions below:

```
$ sudo apt-get update
$ sudo apt-get install build-essential
```

macOS has **not** been tested or adapted. It may not run or reproduce results correctly on macOS. If you try on macOS, please run **without** `+nightly` and **without** `-features="nightly"` first; the build may still fail.

A.2.5 Benchmarks

None.

A.3 Set-up

This project builds on stable Rust by default. The nightly toolchain is **optional** and only required if you want AVX-512 acceleration via the `nightly` feature. Here are the steps to install Rust after installing the build tools in section A.2.4:

1. Install Rust using rustup (the recommended Rust installer):

   ```
   $ curl --proto '=https' --tlsv1.2 -sSf https://
       sh.rustup.rs | sh
   ```

   On Windows, one can download and run the installer "rustup-init.exe" from https://rust-lang.org/tools/install/.

2. After installation, verify Rust is installed correctly:

   ```
   $ rustc --version
   $ cargo --version
   ```

3. (Optional) Install the nightly toolchain (only needed for AVX-512 support):

```
$ rustup toolchain install nightly
```

4. (Optional) Verify the nightly toolchain is available:

```
$ rustc +nightly --version
```

For more information, see the Rust installation guide.

A.3.1 Installation

1. Visit https://doi.org/10.5281/zenodo.17959835 and download the artifacts `tfhe-omr.zip`.

2. Unarchive the `tfhe-omr.zip` file on your local computer.

3. Run the following commands `cd tfhe-omr`.

A.3.2 Basic Test

Remark. If `+nightly` fails in your environment, please use the stable commands (without `+nightly` and without `-features="nightly"`). `+nightly` is optional for AVX-512 acceleration and may not build if toolchain/CPU feature flags do not match; then, stable commands are sufficient and recommended.

1. Build the InstantOMR example.

```
$ cargo +nightly build --package omr_core --
    example omr --features="nightly" --release
```

2. Run InstantOMR example with single thread and single payload.

```
$ cargo +nightly run --package omr_core --
    example omr --features="nightly" --release
    -- --thread-count 1 --payload-count 1
```

3. Run InstantOMR example with 4 threads and 16 payloads.

```
$ cargo +nightly run --package omr_core --
    example omr --features="nightly" --release
    -- --thread-count 4 --payload-count 16
```

It should be noted that the maximum payload-count supported by InstantOMR example is 65536, and the maximum thread-count is recommended to be the number of cores of the machine. For example, a CPU with 8 cores and 16 threads can run 16 threads at the same time, but the optimal operating efficiency is achieved at 8 threads.

A.4 Evaluation workflow

A.4.1 Major Claims

Our major claims about our code are its server (detector) runtime, including its latency, throughput, and parallelizability, using our environment Google Compute Cloud "c3d-highcpu-360", equipped with 180 cores, 708GB of memory, and AVX-512 instructions. In particular:

(C1): Latency: with a single message, InstantOMR server finishes in ∼274 milliseconds.

(C2): Throughput: with 2^{16} messages, InstantOMR server finishes in ∼5 hours

(C3): Parallelizability: with c messages, for c being the number of cores in the server's machine, InstantOMR server finishes in ∼274 ms (nearly identical to the runtime of a single message).

A.4.2 Experiments

(E1): *[Latency] [10 human-minutes + 5 compute-seconds]: Run* InstantOMR *example with single thread and single payload.*

Preparation: *Build the* InstantOMR *example.*

```
$ cargo +nightly build --package omr_core --
    example omr --features="nightly" --release
```

Execution: *Run* InstantOMR *example with single thread and single payload.*

```
$ cargo +nightly run --package omr_core --
    example omr --features="nightly" --release
    -- --thread-count 1 --payload-count 1
```

Results: *The detect time* ∼ 243*ms of the example result shows* InstantOMR *has a low latency described in the paper.*

```
num threads: 1
all payloads count: 1
2025-04-14T14:08:22.064673Z DEBUG ThreadId(01)
    omr: Generating secret key pack...
2025-04-14T14:08:22.065688Z DEBUG ThreadId(01)
    omr: Generating sender and detector...
2025-04-14T14:08:22.719183Z DEBUG ThreadId(01)
    omr: Generating clues...
2025-04-14T14:08:22.720238Z INFO ThreadId(01)
    omr: gen clues time: 945.4us
2025-04-14T14:08:22.720301Z DEBUG ThreadId(01)
    omr: Generating payloads...
2025-04-14T14:08:22.720346Z INFO ThreadId(01)
    omr: gen payloads time: 7.9us
2025-04-14T14:08:22.720628Z DEBUG ThreadId(01)
    omr: Detecting...
[elapsed: 00:00:00] [########################]
    1/1 [eta: 00:00:00] [0s]
2025-04-14T14:08:22.964331Z DEBUG ThreadId(01)
    omr: Detect done
```

```
2025-04-14T14:08:22.964418Z INFO ThreadId(01)
    omr: detect time: 243.6431ms
2025-04-14T14:08:22.964466Z INFO ThreadId(01)
    omr: detect time per message: 243.6431ms
2025-04-14T14:08:22.964819Z INFO ThreadId(01)
    omr: encode indices times: 146.3us
2025-04-14T14:08:22.964878Z INFO ThreadId(01)
    omr: encode indices times per ciphertext:
    48.766us
2025-04-14T14:08:22.965031Z INFO ThreadId(01)
    omr: encode pertinent payloads time: 79.5us
2025-04-14T14:08:23.011517Z INFO ThreadId(01)
    omr: decode time: 46.4251ms
2025-04-14T14:08:23.011637Z INFO ThreadId(01)
    omr: All done
```

(E2): *[Throughput] [10 human-minutes + 5 compute-hours]: Run* InstantOMR *example with single thread and 65536 payloads*

Preparation: *Build the* InstantOMR *example.*

```
$ cargo +nightly build --package omr_core --
    example omr --features="nightly" --release
```

Execution: *Run* InstantOMR *example with single thread and 65536 payloads.*

```
$ cargo +nightly run --package omr_core --
    example omr --features="nightly" --release
    -- --thread-count 1 --payload-count 65536
```

Results: *The detect time* $\sim$ 15340*s of the example result shows the throughput of* InstantOMR*.*

```
num threads: 1
all payloads count: 65536
2025-04-14T14:10:57.838257Z DEBUG ThreadId(01)
    omr: Generating secret key pack...
2025-04-14T14:10:57.838995Z DEBUG ThreadId(01)
    omr: Generating sender and detector...
2025-04-14T14:10:58.444275Z DEBUG ThreadId(01)
    omr: Generating clues...
2025-04-14T14:11:57.382862Z INFO ThreadId(01)
    omr: gen clues time: 58.9381541s
2025-04-14T14:11:57.383002Z DEBUG ThreadId(01)
    omr: Generating payloads...
2025-04-14T14:11:57.403619Z INFO ThreadId(01)
    omr: gen payloads time: 20.5511ms
2025-04-14T14:11:57.403832Z DEBUG ThreadId(01)
    omr: Detecting...
[elapsed: 04:15:40] [########################]
    65,536/65,536 [eta: 00:00:00] [0s]
2025-04-14T18:27:37.612223Z DEBUG ThreadId(01)
    omr: Detect done
2025-04-14T18:27:37.612318Z INFO ThreadId(01)
    omr: detect time: 15340.2083335s
2025-04-14T18:27:37.612369Z INFO ThreadId(01)
    omr: detect time per message: 234.073003ms
2025-04-14T18:27:41.094382Z INFO ThreadId(01)
    omr: encode indices times: 3.4819539s
2025-04-14T18:27:41.094491Z INFO ThreadId(01)
    omr: encode indices times per ciphertext:
    696.39078ms
2025-04-14T18:28:05.354507Z INFO ThreadId(01)
    omr: encode pertinent payloads time:
    24.2598764s
2025-04-14T18:28:05.660156Z INFO ThreadId(01)
    omr: decode time: 305.5313ms
2025-04-14T18:28:05.660291Z INFO ThreadId(01)
    omr: All done
```

(E3): *[Parallelizability] [10 human-minutes + 10 compute-minutes]: Run the* InstantOMR *example with the number of threads equal to the number of payloads.*

Preparation: *Build the* InstantOMR *example.*

```
$ cargo +nightly build --package omr_core --
    example omr --features="nightly" --release
```

Execution: *Run* InstantOMR *example with 8 threads and 8 payloads.*

```
$ cargo +nightly run --package omr_core --
    example omr --features="nightly" --release
    -- --thread-count 8 --payload-count 8
```

Results: *The detect time of this example result is close to the latency got in the first Experiment* **E1**.

```
num threads: 8
all payloads count: 8
2025-12-17T10:37:53.263970Z DEBUG ThreadId(01)
    omr: Generating secret key pack...
2025-12-17T10:37:53.264594Z DEBUG ThreadId(01)
    omr: Generating sender and detector...
2025-12-17T10:37:53.875357Z DEBUG ThreadId(01)
    omr: Generating clues...
2025-12-17T10:37:53.877243Z INFO ThreadId(01)
    omr: gen clues time: 1.6446ms
2025-12-17T10:37:53.877342Z DEBUG ThreadId(01)
    omr: Generating payloads...
2025-12-17T10:37:53.877468Z INFO ThreadId(01)
    omr: gen payloads time: 35.7us
2025-12-17T10:37:53.877592Z DEBUG ThreadId(01)
    omr: Detecting...
[elapsed: 00:00:00] [########################]
    8/8 [eta: 00:00:00] [0s]
2025-12-17T10:37:54.158482Z DEBUG ThreadId(01)
    omr: Detect done
2025-12-17T10:37:54.158596Z INFO ThreadId(01)
    omr: detect time: 280.8241ms
2025-12-17T10:37:54.158686Z INFO ThreadId(01)
    omr: detect time per message: 35.103012ms
2025-12-17T10:37:54.159029Z INFO ThreadId(01)
    omr: encode indices times: 240.7us
2025-12-17T10:37:54.159150Z INFO ThreadId(01)
    omr: encode indices times per ciphertext:
    80.233us
2025-12-17T10:37:54.159574Z INFO ThreadId(01)
    omr: encode pertinent payloads time: 323.9
    us
```

```
2025-12-17T10:37:54.241155Z INFO ThreadId(01)
    omr: decode time: 81.4913ms
2025-12-17T10:37:54.241324Z INFO ThreadId(01)
    omr: All done
```

A.5 Notes on Reusability

We provide an alternative access on our GitHub repository https://github.com/xiangxiecrypto/tfhe-omr, which may contain updated code or instructions.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Message Injection Attacks Against Signal

Kien Tuong Truong
ETH Zurich

Noemi Terzo
Max Planck Institute for Security and Privacy

Kenneth G. Paterson
ETH Zurich

A Artifact Appendix

A.1 Abstract

This artifact consists of a self-contained QEMU virtual machine that allows running the attacks presented in the paper against a victim Signal account. The attacks are both in the malicious server setting and allow an adversary to inject messages in a conversation between two honest, uncompromised users. We reproduce this threat model by running an Android emulator (via nested virtualization), installing the Android Signal application on it (with some modifications, see below), and proxying the connection via *mitmproxy*. This allows message injection from the server into the client running on the emulator. The VM is accompanied by a set of scripts that allow running the attacks and verifying its behaviour.

We depict the setup in Fig. 1.

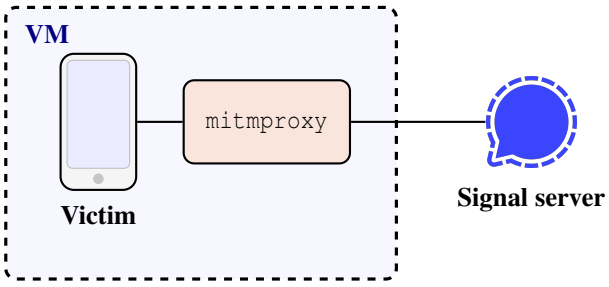


Figure 1: The setup from our artifact. The emulator communicates with the real Signal server via our proxy. Both emulator and proxy are running on the VM. The proxy has message injection capabilities that we can use from within the VM.

A.2 Description & Requirements

For the remainder of this appendix, we will refer to the Signal account running on the emulator as "the victim".

The attacks require two changes to the Signal application on the emulator. It must accept a custom CA certificate for `mitmproxy`, and it must no longer contain the fixes that Signal deployed against our attacks. We apply both automatically during the build. Unfortunately, this means that this process may not work with future versions of the Signal application, as the patch may not apply cleanly. This is fundamentally a constraint given by the fact that the Signal server does not allow connections from clients which are not up-to-date, and therefore we cannot roll back to an unpatched version. Here, we describe all the patches that we apply to the Signal application during the build process, so that future evaluators can adapt them to future versions of the Signal application.

1. Re-enables receiving messages from a PNI address (for attack 1)
2. Re-enables receiving plaintext messages wrapped in a sealed sender envelope (for attack 2)
3. Add a trust root for the sealed sender certificate (for attack 2)
4. Modify the websocket connection to the Signal server to go through `mitmproxy` (for both attacks)
5. Trust the `mitmproxy` CA certificate (for both attacks)
6. Use a fixed secret for the application database (as a convenience feature, not strictly needed for the attacks)
7. Drop the `FLAG_SECURE` flag on the application window (as a convenience feature, not strictly needed for the attacks). This allows seeing the screen of the emulator via `ws-scrcpy`.

Note that the first three items simply revert the patches that Signal has introduced to fix our attacks. A malicious server already has the ability to intercept the connection (so already implicitly has items 4 and 5), and items 6 and 7 are simply convenience features that do not affect the security of the application. We have tested that all patches work for Signal-Android version 8.16.1.

A.2.1 Security, privacy, and ethical concerns

In the setup of our artifact, the victim has to connect to the production Signal server. This demands care when running the attack, as it may negatively affect Signal as a service provider. We have taken care to ensure that our scripts only inject in one direction – from the server to the client – such that no arbitrarily crafted messages are sent to the real server.

Our attacks allow message injection on a victim user. In particular, our second attack allows injecting messages from arbitrary users, given only their username. This raises two ethical concerns. The first one is that one may use our tools to maliciously craft chats with specific users in order to mislead others, or spread misinformation. We believe that such a concern is mitigated by the fact that this is possible regardless of our attack, as a well-motivated attacker may perform the same feat by simply editing their own message database. We do acknowledge that our artifact makes that easier, nonetheless. The second concern is that any reply to our injected messages do hit the Signal server. However, in this case, the reply messages are well-formed from the point of view of the Signal server as they have been created by an honest party (i.e. the victim).

A.2.2 How to access

The latest version of our artifacts is available at https://doi.org/10.6084/m9.figshare.31272589.

For the convenience of the artifact evaluation reviewers, we separately provided access to a zip file (on HotCRP), containing the application state of a Signal user. This allowed testing our attacks without needing two phone numbers and without needing to register a Signal account on the emulator. We did not include this zip file on FigShare as it is not strictly needed for running the attack and because malicious parties may misuse this account if it were to be openly available.

A.2.3 Hardware dependencies

This artifact requires no special hardware to execute. That said, we run a VM with a nested Android emulator, therefore systems that do not support nested KVM virtualization may be too slow for proper usage. Additionally, we build the Signal application within the VM. The build needs memory and CPU power, so the VM claims 16 GiB of RAM and 8 cores by default. A smaller host can shrink it: `RAM_MB=12288 CPUS=6 ./vm/run-vm.sh up` boots the VM with 12 GiB and 6 cores instead, at the cost of a longer build and a slower emulator. The VM disk is a 64 GiB sparse overlay, of which a full build fills roughly 20 GiB.

The first attack also requires another registered Signal account running on another device (presumably a phone). Note that this device will not be negatively impacted by this attack, as it will only send and receive honestly-crafted messages.

A.2.4 Software dependencies

Our artifact requires curl, QEMU, and a cloud-init seed builder to be installed on the host. When the VM is ready, it opens an SSH server on port 2222 to allow access.

On an Ubuntu host, all of these can be installed via:

```
apt install qemu-system-x86 \
            qemu-utils \
            cloud-image-utils \
            openssh-client \
            curl
```

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

After installing the software dependencies mentioned above, ensure that the current user on the host is in the `kvm` group. [1]

Then, download the artifact from the FigShare link provided above and extract it to a folder of your choice. Next, start the VM: `./vm/run-vm.sh up`

This will start the VM and open an SSH server on port 2222. The user account is `ubuntu` and the password is `artefact`.

The cloud-init script will automatically provision all dependencies. Run `cloud-init status --wait` to wait for the provisioning to finish.

Finally, run `SIGNAL_REF=v8.16.1 artefact all`. This clones the Signal Android application at the version we tested and builds it. It also prepares the `mitmproxy` configuration and builds the binaries needed for the attacks. Finally, it creates the emulator and installs the Signal application on it. Note that old Signal client are regularly phased off, so this version may no longer be accepted by the Signal server in the future. Point `SIGNAL_REF` at a newer tag to try a later version. Unfortunately, the patches may no longer apply to newer versions, so this may require some manual work. Note that new releases may need a newer JDK runtime as well.

The whole process takes around 30 minutes.

A.3.2 Basic Test

Run `bash /opt/artefact/repl.sh`. This opens a REPL for running the attack.

Run `doctor` first. It prints one line per tool that the harness needs and ends with `all prerequisites satisfied`.

Then run the `emulators`, `stream` and `mitm` commands in sequence. `emulators` boots the victim emulator and installs the patched application. `stream` starts a ws-scrcpy web server, accessible on the host via http://localhost:8000. `mitm` starts the `mitmproxy` instance and routes the emulator through it.

At this point, the emulator should be usable via http://localhost:8000. Confirm that the Signal application is installed.

[1] If not, add the user to the group via `sudo usermod -aG kvm $USER` and log out and back in for the change to take effect.

A.3.3 (Optional) Signal Account Provisioning

If available, you can use the provided account backup zip file to restore a Signal account on the emulator.

If using the VM, the backup has to be installed on the guest. Place the zip in `backups/` on the host before the first boot, and provisioning copies it in. If the VM is already running, copy it over directly:

```
scp -P 2222 victim-signal-db.zip \
    ubuntu@localhost:/opt/artefact/backups
```

Then run the `restore` command in the REPL and select the zip file. The password was provided to the reviewers via HotCRP. This will restore the account and allow you to run the attacks without needing to register.

Only one copy of the restored account may be online at a time. When a second copy connects, the Signal server closes the older websocket with code 4409 (`Connected elsewhere`) and that client stops receiving messages. Run `restore` again to recover.

A.3.4 Victim Account Extraction

Once the victim account is in place, whether restored or registered by hand, run the `extract` command in the REPL. It reads the victim's databases, prints its ACI and PNI, and writes `attack-script/config.json`. Attack 1 reads that file, so (E1) requires this step. Note that this step only extracts information that a malicious server already has access to, since it is the one that assigns the ACI and PNI addresses to clients.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): A malicious server can inject messages by exploiting the ACI-to-PNI protocol in Signal. This leads to a single "Safety number changed" message, without actually affecting the safety numbers (e.g. if the users compare their QR codes out-of-band, they will succeed). This is the attack described in Section 3 of the paper. This is proven by experiment (E1).

(C2): A malicious server can inject messages by exploiting a weakness in Signal Sealed Sender (SSS) and a validation error for Plaintext messages wrapped using SSS. No "Safety number changed" message appears and safety numbers are not affected. This is the attack described in Section 4 of the paper. Experiment (E2) proves this for a one-on-one conversation. The attack can be extended to groups, though this artifact does not contain the tooling to do so.

A.4.2 Experiments

Inbetween the experiments, it may be convenient to restore the Signal application state to a clean state. If you have access to the provided account backup zip file, you can run the `restore` command in the REPL. Alternatively, create such a backup by using the `backup <name>` command in the REPL and then use the `restore` command.

(E1): *[Attack 1] [10 human-minutes]*

In the following description, we refer to the victim as Alice and to the other device as Bob. Some of the following actions will require interaction from Bob.

Preparation: Set up the environment up to and including the steps described in Section A.3.2. Make sure to have a browser open on the host with access to `http://localhost:8000` and a REPL running. If not using the provided account backup zip file, register a Signal account on the emulator. Either way, run the `extract` command once Alice's account is in place: Attack 1 reads the `config.json` that it writes. Ensure that Bob is also registered on Signal. Do not exchange messages between Alice and Bob before running the attack.

Execution: The following steps run the attack and verify its effects.

1. From Alice's device in the emulator, open Signal and open a new chat (using the pencil icon on the bottom right) with Bob *by using the "Find by phone number" option*.
2. Send a message to Bob.
3. On Bob's device, accept the message from Alice and send a message to Alice. At this point, Alice's device has linked the ACI and PNI addresses of Bob. This fulfils the preconditions for the attack.
4. Run the `attack1` command in the REPL.
5. Select the message to be injected and use Bob's PNI address as the sender. This will inject the message from the server to Alice's device, appearing as if it were sent by Bob. This will also trigger a "Safety number changed" message on Alice's device, but the safety numbers will not actually change.
6. On both devices, click on the name of the contact and press on "View safety number". Confirm that the safety numbers match on both devices.
7. Optionally, send more messages using the REPL. These messages will not trigger any new "Safety number changed" message.
8. Optionally, send messages from Bob to Alice and confirm that they are received correctly, without triggering any new "Safety number changed" message.

(E2): *[Attack 2] [5 human-minutes]*

In the following description, we refer to the victim as Alice. Select a Signal user with a registered username and refer to them as Bob. This attack does not require any

interaction from Bob. For ethical reasons, we strongly recommend using an account under the evaluator's control for Bob.

Preparation: Set up the environment up to and including the steps described in Section A.3.2. Make sure to have a browser open on the host with access to `http://localhost:8000` and a REPL running. If not using the provided account backup zip file, register a Signal account on the emulator. Messages may be exchanged between Alice and Bob prior to the attack.

Execution: The following steps run the attack and verify its effects.

1. Run the `attack2` command in the REPL.
2. Select the message to be injected and use Bob's username as the sender. This will inject the message from the server to Alice's device, appearing as if it were sent by Bob. No "Safety number changed" message will appear, and safety numbers will not be affected.
3. Optionally, send more messages using the REPL. These messages will not trigger any "Safety number changed" message.
4. Optionally, send messages from Bob to Alice and confirm that they are received correctly, without triggering any new "Safety number changed" message.

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/usenixsec2026/`.



USENIX Security '26 Artifact Appendix: Cryptanalysis of LDPC-Based Pseudorandom Error-Correcting Codes

Tianrui Wang[1] Anyu Wang[2,3,4†] Tianshuo Cong[5,6†]
Delong Ran[1] Jinyuan Liu[2] Xiaoyun Wang[2,3,4,7,8]
[1]*Institute for Network Sciences and Cyberspace, BNRist, Tsinghua University*
[2]*Institute for Advanced Study, Tsinghua University* [3]*Zhongguancun Laboratory, Beijing, China*
[4]*State Key Laboratory of Cryptography and Digital Economy Security, Tsinghua University*
[5]*School of Cryptologic Science and Engineering, Shandong University*
[6]*Shandong Key Laboratory of Artificial Intelligence Security, Shandong University*
[7]*National Financial Cryptography Research Center, Beijing, China*
[8]*Shandong Institute of Blockchain, Shandong, China*

A Artifact Appendix

A.1 Abstract

Pseudorandom error-correcting codes (PRCs), a novel cryptographic primitive recently proposed at CRYPTO 2024, are primarily applied in undetectable watermarking schemes for large generative models. However, the security of PRCs has not yet been systematically analyzed. To fill this gap, we present the first cryptanalysis of PRCs. Specifically, focusing on LDPC-PRC, the only known practical instantiation of PRCs, we propose three novel attacks that challenge its undetectability (Attack I, II) and robustness (Attack III). To rigorously demonstrate the practical threat, this artifact analyzes the concrete attack complexity under realistic parameters and validates the attack effectiveness on both real-world large language models and generative image models, including DeepSeek and Stable Diffusion. Our analysis shows that the claimed security guarantees of LDPC-PRC are undermined across all practically feasible regimes.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact attacks the PRC watermark scheme deployed in self-hosted AIGC systems without involving any local data, real-world systems or human subjects. Therefore, it does not raise security, privacy, or ethical concerns.

A.2.2 How to access

Our artifact is permanently and publicly available at https://zenodo.org/records/21771706 with DOI https://doi.org/10.5281/zenodo.21771706.

Our artifact can also be downloaded from GitHub with:
`git clone https://github.com/1234wangtr/PRC_estimator`

A.2.3 Hardware dependencies

Fully running this artifact requires a GPU machine with at least 80GB VRAM (Ours: NVIDIA H20 with 141GB of VRAM), equipped with a CPU with moderate computational power (Ours: Intel XEON PLATINUM 8558P 2.70 GHz), 16GB RAM (Ours: 2TB), and 60GB free disk space. However, the main claims can be verified on a CPU-only machine with 5 GB of free disk space, given pre-generated data.

A.2.4 Software dependencies

OS: Modern x86_64 Linux with `git`, `bash`, and `unzip` installed (Ours: Ubuntu 24.04.1, Kernel 6.17.0-29-generic).
Python: 3.11, with `pip` installed and supports `conda` virtual environments. (Ours: 3.11.4 with Miniconda 25.3.1)
(Optional for GPU) NVIDIA Driver: 525.60.13+ to support CUDA 12.x. (Ours: 595.71.05, CUDA 12.1.105)
(Optional) Fonts: *Times New Roman* and *Droid Sans Fallback*. You can install it via `setup/get_fonts.sh`.

A.2.5 Benchmarks

None. This artifact does not involve third-party benchmarks.

A.3 Set-up

A.3.1 Installation

After downloading the artifact, you can enter `PRC_estimator` folder, which will be the working directory for all the follow-

ing commands.

Our artifact involves two separate Python virtual environments for LLM and GIM experiments. You can set them up with: `conda env create -f llm/environment.yml && conda env create -f gim/environment.yml`

A.3.2 Basic Test

You can run Experiment 1 (E1) to test if the environments are correctly set up.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our attacks reveal that PRC-based watermarking schemes failed to achieve 128-bit security under realistic parameters. This is proven by the experiment E1 described in Section 5.2 of our paper, whose results are presented in Figure 4 and Tables 6 and 7 of our paper.

(C2): Our Attacks I and II can successfully distinguish real-world LDPC-PRC watermarks for generated text. This is proven by the experiment E2 described in Section 6.2.2 and Section 6.2.3 of our paper, whose results are presented in Table 3 of our paper.

(C3): Employing LDPC-PRC watermarks for text generation only works for large temperatures, which leads to a significant quality degradation.

(C4): Our Attacks I and II can successfully distinguish real-world LDPC-PRC watermarks for generated images. This is proven by the experiment E4 described in Section 6.3.2 of our paper, whose results are presented in Table 4 of our paper. These attacks can success even when the adversary has access only to a proxy model. This is proven by the experiment E4 described in Appendix D of our paper, whose results are presented in Table 8 of our paper.

(C5): The attacker can perform Attack III to remove the LDPC-PRC watermarks for images with a high success rate under a reasonable image quality budget. This is proven by the experiment E5 described in Section 6.3.2 of our paper, whose results are presented at the end of Section 6.3.2 of our paper.

A.4.2 Experiments

(E1): Concrete Time Complexity Analysis [1 human-minute + 1 CPU minute]. Estimate the concrete time complexity of our attacks against the LDPC-PRC schemes with LLM and GIM watermarking parameters.
Execution: Run the security estimation scripts as:

```
conda run -n prc-estimator-llm \
  python llm/security_estim/main.py
conda run -n prc-estimator-gim \
  python gim/security_estim/main.py
```

Results: The figures of attack time complexity are in `<llm or gim>/data/security_estim.pdf`, corresponding to Figure 4 (a) and (b). The detailed results are in `<llm or gim>/data/security_estim.csv`, corresponding to Tables 6 and 7.

(E2): Attack I and II against LDPC-PRC Watermarked Text [1 human-minute + 3 CPU-minutes/10 groups]. Perform Attacks I and II against the LDPC-PRC watermarked text generated by DeepSeek-R1-Distill-Qwen-7B under $t = 3$ and temperature 1.8.
Using pre-generated victim data: Since generating watermarked text requires significant GPU resources, we provide 10 groups of pre-generated watermarked text under $t = 3$. You can obtain the data by:

```
unzip llm/data/Deepseek_t_3_temp_1.8.zip \
  -d llm/data/Deepseek_t_3_temp_1.8_example
```

(Optional) Self-generating victim data: [Extra 5 GPU-hours and 0.2GB disk space for 10 groups, extra 16GB disk space for model] First, download the model using `setup/get_llm.sh`. Then, generate watermarked text with:

```
conda activate prc-estimator-llm
export CUDA_VISIBLE_DEVICES=0
python llm/generation/main.py \
  --temperature 1.8  --model_name Deepseek \
  --prc_t 3 --start 0 --end 10
```

This will generate 10 groups of watermarked text with different pk to `llm/data/Deepseek_t_3_temp_1.8`.
Execution: The attacks can be performed with:

```
conda activate prc-estimator-llm
python llm/attack/attack1_2_main.py --t 3 \
  llm/data/Deepseek_t_3_temp_1.8[_example]
```

This attack aims to distinguish PRC codewords and random vectors.
Results: The attack results are in `llm/data/Deepseek_t_3_temp_1.8[_example]/result.csv`, corresponding to Table 3. The meaning of each field is as follows:

- **TPR_0:** The true-positive rate at which PRC codewords are correctly classified (higher is better).
- **FPR_0:** The false-positive rate at which random vectors are incorrectly classified as PRC codewords (lower is better).
- **TPR_1:** Measures whether the inversed PRC codeword can be detected as a watermark. A significant gap between TPR_1 and FPR_0 demonstrates better effectiveness of the attack.
- **Success Rate:** For Attack-I, it corresponds to the proportion of instances in which at least one row of the secret key is successfully recovered. For Attack-II, it corresponds to the proportion of instances in which at least one pair of identical rows is successfully identified.

Although our paper reports results over 128 groups, the results over 10 groups should have the same trend.
The attack details are in `llm/data/Deepseek_t_3_temp_1.8[_example]/log.txt`. You can inspect

some intermediate outputs such as:

```
# Recovered nonzero positions of the secret key
grep "found l1_vec" \
llm/data/Deepseek_t_3_temp_1.8[_example]/log.txt
# Duplicated positions identified from pub key
grep "dup_dict" \
llm/data/Deepseek_t_3_temp_1.8[_example]/log.txt
```

(E3): Watermarked Text Quality Analysis [2 human-minutes + 2 CPU minutes]. Analyze the pre-generated quality of the generated watermarked text.

Preparation: Since the quality analysis requires lots of watermarked text generated under different temperatures, the analysis is performed on the pre-generated data, which can be obtained by:

```
unzip -d llm/data \
  llm/data/Deepseek_t_3_temp_all.zip
```

Execution: Run the following command to analyze the quality of the generated watermarked text.

```
conda run -n prc-estimator-llm \
  python llm/generation/plot_entropy.py
```

Results: The generated figure is in `llm/data/avg_entropy_vs_correct_rate.pdf`, corresponding to Figure 5.

(E4): Attack I and II against LDPC-PRC Watermarked Image [1 human-minute + 6 CPU-minutes/10 groups]. Perform Attacks I and II against the LDPC-PRC watermarked image generated under $t = 3$.

Using pre-generated victim data: Since generating watermarked images requires significant GPU resources, we provide 10 groups of pre-generated watermarked images under $t = 3$. You can obtain the data by:

```
unzip gim/data/SD21_t3.zip \
  -d gim/data/SD21_t3_example
unzip gim/data/SD21_t3_inv_SD15_SD2_SD21.zip \
  -d gim/data/SD21_t3_inv_SD15_SD2_SD21_example
```

(Optional) Self-generating victim data: [Extra 2 GPU-hour, 1GB disk space for 10 groups, extra 25GB disk space for model] First, download the model and dataset with `setup/get_gim.sh`. Then, generate watermarked images with:

```
conda activate prc-estimator-gim
export CUDA_VISIBLE_DEVICES=0
python gim/generation/main.py \
  --prc_t 3 --start 0 --end 10 \
  --gen_model_id SD21 --inv_model_ids SD21
# using proxy models
python gim/generation/main.py \
  --start 0 --end 10 --gen_model_id SD21 \
  --inv_model_ids SD15,SD2,SD21 --prc_t 3
```

This will generate 10 groups of watermarked images with different pk to `gim/data/SD21_t3`, along with 10 groups of watermarked images inverted by 3 different proxy models to `gim/data/SD21_t3_inv_SD15_SD2_SD21`

Execution: The attacks can be performed with:

```
conda activate prc-estimator-gim
python gim/attack/attack1_2_main.py --t 3 \
  gim/data/SD21_t3[_example]
python gim/attack/attack1_2_diff_inv_main.py \
  gim/data/SD21_t3_inv_SD15_SD2_SD21[_example] \
  --t 3 # using proxy models
```

This attack aims to distinguish PRC codewords and random vectors.

Results: The attack results without using proxy model are in `gim/data/SD21_t3[_example]/result.csv`, corresponding to Table 4. The attack results with proxy model are in `gim/data/SD21_t3_inv_SD15_SD2_SD21[_example]/result.csv`, corresponding to Table 8. The meaning of these results are the same as described in (E2). Although our paper reports results over 128 groups, the results over 10 groups should have the same trend. You can inspect some intermediate outputs at the corresponding `log.txt` files as demonstrated in (E2).

(E5): Watermark Removal Analysis [5 human-minutes + 2 GPU-hour for 10 images].

Preparation: Make sure the model and dataset are downloaded with `setup/get_gim.sh`.

Execution: First, generate watermarked images with:

```
conda activate prc-estimator-gim
export CUDA_VISIBLE_DEVICES=0
python gim/generation/main-for-attack3.py \
  --prc_t 3 --start 0 --end 10 \
  --model_id SD21
```

This will generate 10 watermarked images and save intermediate results to `gim/data/attack3_SD21_t3` in 2 GPU-minutes.

Then, generate the adversarial images with $\varepsilon = 16.0$ based on those intermediate results (corresponding to the attack results for Attack III) with:

```
conda activate prc-estimator-gim
export CUDA_VISIBLE_DEVICES=0
python gim/attack/attack3_main.py \
  gim/data/attack3_SD21_t3 --start 0 --end 10 \
  --model_id SD21 --eps 16.0
```

The attack will run for 1 GPU-hour for 10 images.

Results: The generated adversarial images are in `gim/data/attack3_SD21_t3/inv_lat_16.0`, and the images with removed watermarks are in `gim/data/attack3_SD21_t3/adv_img_16.0`.

To calculate the removal success rate, run:

```
conda activate prc-estimator-gim
python gim/attack/attack3_stati.py \
  gim/data/attack3_SD21_t3/inv_lat_16.0
```

This result corresponds to the attack success rate reported at the end of Section 6.3.2.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: On Improving Robustness of Deepfake Image Detectors

Abu Taib Mohammed Shahjahan
Concordia University, Montreal, Canada

Mohammad Mannan
Concordia University, Montreal, Canada

Abdessamad Ben Hamza
Concordia University, Montreal, Canada

Amr Youssef
Concordia University, Montreal, Canada

A Artifact Appendix

A.1 Abstract

This artifact constitutes the official codebase and pre-trained model weights accompanying the USENIX Security 2026 paper "On Improving Robustness of Deepfake Image Detectors." The artifact provides a complete software framework implemented in PyTorch for reproducing all major experimental results, including the evaluation of the proposed primary detector, Improved-D3 (`D3_imp`), with the MM-BSN denoising module, alongside five legacy baseline detectors (CNN-F, Resynthesis, DCT, DE-FAKE, and Patch-Forensics). The source code is permanently archived on Zenodo (https://doi.org/10.5281/zenodo.20327567) and actively maintained on GitHub (https://github.com/shahjahan0275/semantic_adv).

To accommodate varying resources, we provide two evaluation pathways: a **Functional Setup** using `sample_data` and pre-trained checkpoints for quick verification, and a **Full Reproduction Setup** requiring the ∼623 GB GenImage and adversarial datasets to fully replicate Tables 1–4 in our paper. Dependencies are strictly managed via two Anaconda environments: Python 3.10 for the primary detector (`D3_imp`), and Python 3.8 with PyTorch 1.12.1 for legacy baselines. The provided scripts, SLURM files, and detailed `README.md` facilitate execution across consumer GPUs (for legacy functional tests) and HPC H100/A100 GPUs (mandatory for all training and primary model inference) to seamlessly validate functionality, robustness metrics, and training convergence.

A.2 Description & Requirements

We outline the hardware and software requirements for evaluating our artifact in §A.2.3 and §A.2.4, respectively. The `README.md` file in our artifact includes step-by-step instructions to help users configure their environments. To facilitate evaluation, we provide two distinct evaluation pathways based on data requirements. We refer to the first scenario as the "Functional Setup," which utilizes a provided `sample_data` directory and pre-trained checkpoints to allow USENIX Security evaluators to quickly verify the codebase's functionality without downloading hundreds of gigabytes of data (e.g., the 623 GB GenImage dataset). We refer to the second scenario as the "Full Reproduction Setup," which requires downloading the massive, complete external datasets to fully replicate the time-intensive training and evaluation results reported in the paper.

The hardware requirements are dictated by the specific model architectures rather than the evaluation pathway. Due to the integration of the memory-intensive MM-BSN module and the large size of the trained weights (∼1.8 GB), executing both training and inference for our primary `D3_imp` detector strictly requires high-performance computing (HPC) cluster nodes (e.g., NVIDIA H100 or A100), even when utilizing the reduced `sample_data`. Standard consumer-grade GPUs lack the required VRAM for `D3_imp` and are only sufficient for running the functional inference tests of the five legacy baseline detectors (CNN-F, Resynthesis, DCT, DE-FAKE, and Patch-Forensics). The remainder of this section details these dependencies and the basic installation steps.

A.2.1 Security, privacy, and ethical concerns

The artifact does not contain any destructive scripts, malware, or functionality that would compromise the security of the evaluator's machine. All operations are limited to standard deep learning training and inference tasks, including data loading, model training, and evaluation. The codebase does not disable any security mechanisms, collect personally identifiable information, or require network access beyond downloading pre-trained weights and datasets from public sources.

A.2.2 How to access

The official archived artifact corresponding to the USENIX Security 2026 paper is permanently available via Zenodo at the following stable DOI reference: https://doi.org/10.5281/zenodo.20327567. The active development repository containing the source code is hosted on GitHub (https://github.com/shahjahan0275/semantic_adv). Pre-trained

model weights required for inference are hosted externally on Google Drive, with direct access links provided within the artifact's `README.md`.

A.2.3 Hardware dependencies

The hardware requirements strictly depend on the chosen evaluation pathway:

- **Full Reproduction Setup:** Executing the full training and evaluation pipeline requires high-performance GPUs. The primary `D3_imp` detector strictly requires an NVIDIA H100 (96GB VRAM) to prevent CUDA Out-Of-Memory (OOM) errors. The legacy detectors (e.g., CNN-F, Resynthesis, DCT, DE-FAKE, and Patch-Forensics) strictly require an NVIDIA A100 (80GB) GPU. Evaluators must have access to such hardware (80GB–96GB VRAM) to execute any training scripts, as standard local workstations (e.g., RTX 4500) possess insufficient memory for these models. Inference scripts for the legacy detectors (e.g., CNN-F, Resynthesis, DCT, DE-FAKE, and Patch-Forensics) could be run on standard local workstations (e.g., RTX 4500). We do not provide SSH access to our machines; evaluators pursuing full reproduction must provision their own equivalent HPC hardware.
- **Functional Setup:** To verify functionality via the basic tests (§A.3.2) using the provided `sample_data`, we still need NVIDIA A100 (80GB).

A.2.4 Software dependencies

Our framework operates on a standard Linux OS (e.g., Ubuntu) and relies on the PyTorch ecosystem. Strict dependency management is enforced via Anaconda/Miniconda. We provide two distinct Conda configurations to support models spanning different architectural generations:

- **Primary Environment:** Requires Python 3.10 and is used for evaluating our proposed primary `D3_imp` detector.
- **Legacy Defense Environment:** Requires Python 3.8, PyTorch 1.12.1, and CUDA 11.3, specifically configured to run the older detectors (e.g.,CNN-F, Resynthesis, DCT, DE-FAKE, and Patch-Forensics).

We provide complete `environment.yml` files for seamless Conda initialization, alongside `requirements.txt` files as a fallback for standard pip installations.

A.2.5 Benchmarks

For the Full Reproduction Setup, evaluators must download the following external datasets (not included in the artifact due to massive size limitations, the URLs are provided in our `README.md` file):

- **GenImage Dataset:** Used for training the primary model ($\sim$623 GB).
- **Adversarial & Surrogate Datasets (Abdullah et al., IEEE SP 2024):** Used for robustness evaluation across Tables 1–4 (e.g., StyleCLIP adversarial samples).

For the Functional Setup, we have provided a `sample_data` folder within the artifact containing a minimal subset of these benchmark images.

A.3 Set-up

A.3.1 Installation

To prepare the environment, first install Anaconda or Miniconda. Clone the repository and configure the environments using the provided YAML files.

1. Install the Primary Environment:

```
conda env create -f environment.yml
conda activate D3
```

Fallback (if Conda creation fails):

```
conda create -n D3 python=3.10
conda activate D3
pip install -r requirements.txt
```

2. Install the Legacy Environment: Navigate to the `EvolvingThreat-DeepfakeImageDetect/defenses/` directory and repeat the process for the older baselines:

```
conda env create -f environment.yml
conda activate defake
```

3. Download Pre-trained Weights: Follow the Google Drive links in the `README.md` to download the pre-trained checkpoints. Place them in their respective `ckpt/` directories as specified. Update any absolute file paths in the scripts to match your local directory structure.

A.3.2 Basic test

To verify that the primary model environment is correctly configured and functional, you will run a lightweight inference script using the adversarial fake dataset provided by Abdullah et al. (IEEE SP 2024).

Execution:

```
conda activate D3
python validate_for_robustness_spd.py
```

Note: Ensure the dataset paths inside `validate_for_robustness_spd.py` are pointing to the local directory containing any sub-folder of the `sample_data` (e.g., sample_data/StyleCLIP_dataset) or any one of the adversarial fake datasets provided by Abdullah et al. (IEEE SP 2024) and the model path pointing to the downloaded checkpoint.
Expected Output: The script will initialize the model on the GPU without OOM errors, process the small batch of images in the `sample_data` or adversarial fake dataset folder, and output a summary of the detection metrics (e.g., Recall/Accuracy on the sample batch) alongside a `.xlsx` log file detailing the prediction scores for each processed image.

A.3.3 Major claims

The following major claims (C1–C4) are made in the paper and are validated by the experiments (E1–E3) described in the next subsection. These claims are structured to provide a complete evaluation workflow: C1 establishes the baseline problem (the performance degradation that exists in current detectors), C2 and C3 demonstrate the effectiveness of our proposed solutions, and C4 explains why our approach works through ablation analysis. Together, they enable artifact evaluators to verify both the existence of the problem and the efficacy of our proposed improvements using the provided codebase.

(C1): Recent state-of-the-art deepfake detectors (seven models from 2024–2025) suffer severe performance degradation under the content-manipulating adversarial attacks of Abdullah et al. (IEEE SP 2024). For example, accuracy drops from 98.9% to 48.7% for Directionality, and even the best-performing detector D^3 (CVPR 2025) degrades from 86.7% to 81.9%. C1 establishes the **baseline problem** that our work addresses. It provides the "before" comparison against which our improvements (C2, C3) are measured. Without reproducing this degradation, evaluators cannot validate the improvement magnitude claimed in C2 (e.g., 81.9% $\rightarrow$ 97.15%). This claim also justifies why the repository includes code for evaluating seven SOTA detectors beyond our own model, and provides context for the adversarial datasets used throughout the evaluation.

How to reproduce: This claim is demonstrated in Section 3 of the paper (Table 1) and can be reproduced using experiment (E2) on the full adversarial datasets, following the "Reproducing Table 1" instructions in the GitHub `README.md`.

(C2): Our proposed framework, which integrates patch-shuffling & rotation, MM-BSN content-agnostic residual features, and fourth-order DCT-based moment pooling, consistently improves robustness without adversarial training. When integrated into D^3, our method increases accuracy from 81.9% to 97.15% on the StyleCLIP adversarial dataset, with average AP gains of +9.94% (see Table 3 of the paper).

Why this claim is included: C2 demonstrates the primary contribution of our work: significantly improving the best-performing baseline detector (D^3) against adversarial attacks. The 15.25% accuracy gain (81.9% $\rightarrow$ 97.15%) directly shows the effectiveness of our three-component framework. This claim validates that our approach works on a complex, modern architecture without requiring adversarial training data.

How to reproduce: This claim is supported by experiment (E2) using the pre-trained enhanced model, following the "Reproducing Table 3" instructions in the GitHub `README.md`.

(C3): Our modifications substantially reduce recall degradation for five weaker detectors (DCT, DE-FAKE, CNN-F, Patch-Forensics, Resynthesis) across three surrogate models (EfficientNet, ViT, CLIP-ResNet). The maximum reduction is 88.9% (DCT, CLIP-ResNet) and the minimum is 19.5% (Patch-Forensics, CLIP-ResNet); details are in Table 4 of the paper.

Why this claim is included: C3 demonstrates that our approach is **architecture-agnostic** and broadly applicable, not just tailored to D^3. By showing significant improvements across five distinct detectors with different architectural priors, we prove the generalizability of our techniques. The variation in improvement magnitude (19.5%–88.9%) also provides insight into which detector designs benefit most from higher-order statistics.

How to reproduce: This claim is reproduced by experiment (E2) on the adversarial datasets using the retrained models in the `EvolvingThreat-DeepfakeImageDetect/defenses/` folder, following the "Reproducing Table 4" instructions in the GitHub `README.md`.

(C4): Ablation studies (Section 5.3, Table 5 & 6) show that fourth-order DCT statistics (kurtosis) are the critical driver of improvement, outperforming first-, second-, and third-order pooling; patch shuffling alone matches the original D^3 performance, confirming that gains are not from spurious artifacts.

Why this claim is included: C4 explains why our approach works by isolating the contribution of each component. It demonstrates that performance gains are not simply due to increased model capacity or added features, but specifically arise from fourth-order (kurtosis) statistics capturing tail-heavy distributions that adversarial attacks leave unconstrained. This ablation study also confirms that patch shuffling does not introduce spurious discriminative artifacts.

How to reproduce: This claim is validated by experiment (E2) when comparing different pooling orders, and by experiment (E3) for training convergence, using the configurable training scripts provided in the repository (e.g., `train_3branch_FOS_F.py`).

A.3.4 Experiments

The following experiments are designed to validate the functionality of the provided codebase and to reproduce the primary claims regarding detector robustness reported in Tables 1–4. Please note that hardware constraints (H100/A100 vs. consumer GPUs) and data download requirements scale significantly between the "Functional" and "Full Reproduction" setups. We describe the "Functional" under experiment 1 (E1) and "Full Reproduction" are divided into two Experiments experiment 2 (E2) and experiment 3 (E3) for full inference and full training with full dataset as described in the Zenodo/GitHub code repository.

(E1): [Functional Verification via Sample Data] [Approximately 30 human-minutes + 1 compute-hour + 5GB disk]: This experiment verifies the functional correctness of the primary `D3_imp` detector and other five detector models without requiring massive dataset downloads. It utilizes the minimal `sample_data` provided in the repository.
Preparation: Activate the respective conda environments (`D3` or `defake`). Download the pre-trained model weights from the provided Google Drive link and place them in their respective `ckpt` or `checkpoints` directories.
Execution: For the primary `D3_imp` detector, execute `python validate_for_robustness_spd.py` using the sample data. Note: This specific script must be executed on an HPC node equipped with an NVIDIA H100 or A100 to prevent OOM errors during the MM-BSN module initialization. For the five legacy detectors (CNN-F, Resynthesis, DCT, DE-FAKE, Patch-Forensics), execute their respective inference commands from the `README.md` using the `sample_data`; these inferences can be successfully run on standard consumer-grade GPUs.
Results: The models should successfully allocate to the GPU, process the sample images, and terminate without CUDA OOM errors. This successfully demonstrates the end-to-end functional integrity of the inference codebase.

(E2): [Detector Robustness Evaluation (Tables 1, 2, 3 & 4)] [30 human-minutes + 4 compute-hours + 60GB disk]: This experiment reproduces the inference-based robustness evaluations reported in Tables 1, 2, 3 and 4 of the paper, testing the detectors against the full evaluation datasets.
Preparation: Download the evaluation datasets (Section 5.2 of Abdullah et al., IEEE SP 2024). Update the dataset directory paths (e.g., `-fake_root`, `-real_root`, `-dir`) and absolute checkpoint paths within the respective inference scripts to point to your local directories.
Execution: To reproduce Table 1 and 2 please go to the relevant section of the `README.md` file in the code repository. To reproduce Table 3, run `validate_for_robustness_spd.py` using the `D3_imp` weights and all the datasets of Section 5.2 of Abdullah et al. (IEEE SP 2024). To reproduce Table 4, run the batch inference scripts across the legacy detectors located in the `EvolvingThreat-DeepfakeImageDetect/defenses/` folder (e.g., `infer_dct_LM1.py`, `infer_dct.py`, `test_dct.py`) using the three adversarial datasets of Abdullah et al., IEEE SP 2024 (CLIPResNet, EfficientNet, ViT). Optionally, execute `python plot_tsne.py` to generate the t-SNE visualizations (Figures 5 and 6).
Results: The output logs and `.csv` files will contain the evaluation metrics. The calculated results should closely match the robustness metrics reported in Tables 3 and 4 of the manuscript.

(E3): [End-to-End Training] [4 human-hours + 19 compute-days + 750GB disk]: This experiment covers the complete, time-intensive reproduction of the model training phases for both the primary `D3_imp` detector and the legacy baseline defenses.
Preparation: Download the complete GenImage dataset (∼623 GB) and the training datasets from Abdullah et al. (IEEE SP 2024). Ensure the HPC node has at least 750GB of free storage. Activate the relevant environment and update the `-checkpoints_dir`, `-dataroot`, and dataset root flags in the training scripts.
Execution: For `D3_imp`, execute `train_3branch_FOS_F.py` using the parameters for either the StyleCLIP only dataset or the combined StyleCLIP and GenImage datasets. For the legacy models, run their respective training scripts (e.g., `train_dctstats.py`, `train_FOS_3BDCT_patch_text.py`, `train_V1.py`) or submit the provided SLURM scripts (e.g., `Resynthesis_styleclip.sh`) to the cluster scheduler.
Results: The training scripts will pipe outputs to log files (e.g., `logs/train_*.log`) and yield new model checkpoint files (`.pth` or `.pt`) in the specified output directories, demonstrating successful training convergence on the full datasets.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Adversarial Patch EXterminator: Zero-Shot and Patch-Agnostic Defense Framework Against Adversarial Patch Attacks

Jiayimei Wang[1] Tao Ni[2] Guowen Xu[3] Qingchuan Zhao[1] Cong Wang[1]
[1]City University of Hong Kong
[2]King Abdullah University of Science and Technology
[3]University of Electronic Science and Technology of China

A Artifact Appendix

A.1 Abstract

APEX is a zero-shot defense framework designed to mitigate adversarial patch attacks by localizing patch regions and restoring the original image. This artifact enables the reproduction of the main experiments conducted in our paper. Specifically, the artifact provides the implementation of the complete APEX defense pipeline, including bounding-box extraction, patch localization, and image inpainting. Furthermore, the artifact provides a diverse collection of adversarial patch examples, including non-naturalistic, naturalistic, and infrared patches on multiple pedestrian datasets, together with patch location annotations and ground-truth labels.

In addition, the artifact includes configuration files and instructions for reproducing the key experimental results shown in the paper, such as patch localization recall and false positive rates across different patch types. Besides, It also supports the reproducibility of robustness evaluations of APEX under challenging scenarios, including tiny patches, irregularly-shaped patches, background-coherent patches, and adversarial patches placed on abstract paintings.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact does not pose risks to the evaluator's machine security, data privacy, or raise ethical concerns. It does not require disabling any security mechanisms, modifying system-level configurations, or executing destructive operations. In addition, all experiments are conducted on publicly available datasets or synthetic adversarial patches generated for research purposes.

A.2.2 How to access

The artifacts are available at Zenodo: https://zenodo.org/records/19233434. Please make sure to download the latest version (v4 or later).

A.2.3 Hardware dependencies

All experiments in the paper were conducted on NVIDIA RTX A6000 GPUs with 48 GB of GPU memory. However, the artifact can also be executed on other CUDA-capable GPUs with at least 6 GB of GPU memory, as the peak GPU memory usage during execution is approximately 5–6 GB for a single GPU.

A.2.4 Software dependencies

The artifact is implemented in Python (Python 3.10 is recommended) and relies on common deep learning libraries. All required dependencies are specified in `requirements.txt`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

We refer the evaluator to the README for the installation instructions. Specifically, The weights for YOLOv5, YOLOv8, SAM, and LaMa are all already included in the resources provided in Zenodo. Manual installation is only required if any of these four models are missing or corrupted, or if the evaluator wishes to test additional models beyond them.

A.3.2 Basic Test

The artifact includes adversarial patch examples for functionality verification. Therefore, after completing the installation, the evaluator can perform a simple end-to-end test to verify that all required components of the APEX pipeline are functioning correctly. First, ensure the current working directory is the artifact root:

```
cd APEX
```

Clear previous outputs and run the complete pipeline on the provided input images:

```
rm -rf lama/output/* lama/LaMa_test_images/*
python3 auto_pipeline.py \
  --source <path-to-input-image-directory> \
  --output <path-to-output-directory>
```

If the test is successful, the pipeline will execute bounding-box extraction, patch localization, and image inpainting, and the output directory will contain the corresponding intermediate and final results.

Specifically, the output directory directly contains the following subdirectories:

```
<output-directory>/
|-- final_results/
|-- human_crops/
`-- locate_result/
```

final_results/ This directory contains the final visualization results for each input image:

```
final_results/
|-- 00000.jpg
|-- 00000_mask.png
|-- 00000_overlay.png
`-- ...
```

The files are interpreted as follows: `*_mask.png` are binary masks where black regions indicate detected adversarial patch areas; `*_overlay.png` are visualizations with detected masks overlaid and highlighted in red; `*.jpg` are restored images with detected patch regions removed.

human_crops/ This directory contains image crops produced by the bounding-box extraction stage:

```
human_crops/
|-- 00000_crop_0.jpg
|-- 00001_crop_0.jpg
|-- 00001_crop_1.jpg
`-- ...
```

Each image corresponds to a bounding box extracted by the human detector. Therefore, these crops reflect the output of the bounding-box extraction module (`bb_extraction.py`).

locate_result/ This directory stores the patch localization results for each cropped region:

```
locate_result/
|-- 00000_crop_0.png
|-- 00000_crop_0_detection_report.json
|-- 00000_crop_1.png
|-- 00000_crop_1_detection_report.json
`-- ...
```

The `*.png` files visualize the patch localization results for each crop, while the corresponding `*_report.json` files record patch location metadata. These metadata are used to map the localized patches back to their corresponding positions in the original images during the final stage.

In addition, the directory `APEX/lama/output/` contains files named `*_mask.png`, which store images where the black mask regions have been restored using the LaMa inpainting model.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): *Applying APEX improves the detection performance of object detectors under adversarial patch attacks. This is demonstrated by the Overall Defense Performance results reported in § 5.2 (Effectiveness), where detectors achieve higher mAP after applying the APEX defense.*

(C2): *APEX generalizes across multiple types of adversarial patches, including non-naturalistic patches (Non-NAP), naturalistic patches (NAP), and infrared patches (IRP), while maintaining high localization recall and low false positive rates. This claim is supported by the Patch Localization Performance and False Positive Analysis results in § 5.2 (Effectiveness).*

(C3): *APEX remains effective under extreme adversarial scenarios, including tiny patches, irregularly shaped patches, and background-coherent patches. This is evidenced by the experimental results illustrated in Figure 5 and discussed in § 5.2 (Effectiveness).*

(C4): *APEX is capable of detecting adversarial patches even in non-photorealistic visual domains, such as abstract paintings. This claim is demonstrated by the results reported in § 5.4 (Effectiveness in Abstract Paintings).*

A.4.2 Experiments

(E1): *Overall Defense Performance (mAP Evaluation) [15 human-minutes + 1 compute-hour].*

Preparation: *This experiment uses Version 4.1 of the artifact. In practice, to use Version 4.1, one only needs to replace `APEX_mAP_origin` and `collect_apex_results.py` in `APEX_v4` with the updated versions, then copy `yolov8s_finetuned.pt` and `patch_v8s_ft_latest.jpg` into `APEX_v4`. Ensure that the updated dataset `APEX_mAP_origin` (500 images), the fine-tuned model weights (`yolov8s_finetuned.pt`), and the fine-tuned adversarial patch (`patch_v8s_ft_latest.jpg`) are all available under `APEX_v4`.*

Execution: *First, evaluate the detection performance (AP@0.5) before and after applying the patch:*

```
python add_eval.py \
```

```
  --patch   patch_v8s_ft_latest.jpg \
  --dataset APEX_mAP_origin \
  --weights yolov8s_finetuned.pt
```

Next, save all adversarial samples generated in the previous step:

```
python gen_patched_dataset.py \
  --patch   patch_v8s_ft_latest.jpg \
  --dataset APEX_mAP_origin \
  --out-dir APEX_mAP
```

Then, run the defense with:

```
python multigpu_no_lama.py \
  --dataset_dir APEX_mAP \
  --output_dir  APEX_masks \
  --num_gpus <x>
```

To restore a clean dataset:

```
python collect_apex_results.py \
  --masks-dir \
  APEX_masks/apex_output/final_result_merged \
  --original-dataset APEX_mAP \
  --output APEX_mAP_cleaned \
  --lama-dir lama \
  --batch-size 50
```

Finally, evaluate the cleaned dataset by running:

```
python eval.py \
  --dataset APEX_mAP_cleaned \
  --weights yolov8s_finetuned.pt
```

Results: *The experiment reports the AP@0.5 of the detector on adversarially patched images before and after applying the `APEX` defense. Due to the random placement of the adversarial patch, results may fluctuate by approximately 3% across runs. Across three independent runs, the patched AP@0.5 consistently falls in the range **40.4%–40.7%**, and the AP@0.5 after `APEX` defense consistently falls in the range **85.0%–87.8%**, compared to a clean-image AP@0.5 of **90.9%**. These results correspond to the YOLOv8/AdvPatch AP@0.5 after `APEX` defense as reported in Table 2.*

(E2): *Patch Localization Recall and False Positive Evaluation [10 human-minutes + 2 compute-hours].*

Preparation: *Ensure that the directory `APEX_dataset` exists and contains the following three subdirectories corresponding to different patch types: `Infrared_patch`, `Natural_patch`, and `NonNatural_patch`.*

Execution: *Run the following command to evaluate patch localization performance:*

```
python3 test_recall_fpr.py \
  --dataset_dir <path-to-patch-dataset> \
  --output_dir <path-to-output-directory> \
  --num_gpus <number-of-gpus> \
  --iou_threshold 0.5 \
  --patch_class_id 0
```

The argument `-dataset_dir` should be set to the path of each subdirectory under `APEX_dataset` (i.e., one run per patch type). The value of `-num_gpus` should be adjusted according to the available GPUs, and using more GPUs is recommended for improved efficiency. The IoU threshold is fixed to 0.5, consistent with the paper.

Results: *The script reports the total number of processed images, the average localization recall (Table 3), the average false positive rate (Figure 6), and the total number of correctly detected patches in the format `detected/total`. In addition, the patch detection results are saved under the output directory, following the same format illustrated in § A.3.2.*

(E3): *Evaluations on Extreme Cases [10 human-minutes + 1 compute-hour].*

Preparation: *Ensure that the directory `APEX_extreme_case` is available. All other preparation steps are identical to those in Experiment (E2).*

Execution: *Execute the same command as in Experiment (E2), with the argument `-dataset_dir` set to the paths of the three subdirectories under `APEX_extreme_case`, namely `tiny_patch`, `irregular_shape`, and `background_coherent` (one run per subdirectory).*

Results: *The output includes localization recall (Figure 5), false positive rate, and patch detection results saved under the output directory, following the same format as Experiment (E2) and § A.3.2.*

(E4): *Evaluations on Abstract Paintings [10 human-minutes + 30 compute-minutes].*

Preparation: *Ensure that the directory `APEX_abstract` is available. All other preparation steps are identical to those in Experiment (E2).*

Execution: *Execute the following command with `-dataset_dir` set to `APEX_abstract` and `-min_fp_area_ratio` set to `0.01` to match the paper's FPR threshold:*

```
python test_recall_fpr.py \
  --dataset_dir APEX_abstract \
  --output_dir  APEX_APEX_abstract_output \
  --iou_threshold 0.5 \
  --patch_class_id 0 \
  --min_fp_area_ratio 0.01
```

Results: *The output includes localization recall, false positive rate, and patch detection results saved under the output directory (Figure 9), following the same format as Experiment (E2) and § A.3.2.*

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Your Keywords Know Each Other: Breaking SSE with <1% Leaked Documents

Mingyu Bian[1] Jiabei Wang[1, ✉] Dandan Xu[1]
Guangyu Huang[1] Yongbin Zhou[1, ✉]

[1]*School of Cyber Science and Engineering, Nanjing University of Science and Technology, China*
{saunafish, wangjiabei, xudandan, huangguangyu, zhouyongbin}@njust.edu.cn

A Artifact Appendix

A.1 Abstract

The artifact is a Python reference implementation of WHISPER, the leakage-abuse attack on Searchable Symmetric Encryption (SSE) introduced in the paper. It bundles the WHISPER attack and its greedy and late-fusion variants, the VAL and LEAP baselines, the LLM-assisted co-occurrence pipeline, dataset parsers for six public corpora, and ten self-contained scripts that reproduce every figure and table in the paper. The artifact ships pre-built keyword and document-body caches for all six datasets, so the experiments run end-to-end without downloading any raw corpus. All experiments run on a commodity CPU machine; only the optional LLM-synthesis step requires an (inexpensive) third-party API key.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. The artifact performs no destructive operation, contacts no live SSE service, and disables no security mechanism on the evaluator's host. It only reads the shipped public-dataset caches, runs local computation, and writes result logs under `results/`. The optional LLM-synthesis path sends *sanitized* seed text (headers, quoted replies, and contact information removed) to a third-party OpenAI-compatible API using the evaluator's own key; evaluators who prefer not to use an external service can run the structural-only variant (`multiplier=0`), which makes no network calls.

A.2.2 How to access

Permanently archived on Zenodo under the concept DOI https://doi.org/10.5281/zenodo.20355588, which always resolves to the latest published version of the artifact.

✉ Corresponding authors.

A.2.3 Hardware dependencies

No special hardware is required. A commodity x86_64 or Apple-silicon workstation with at least 16 GB of RAM and a few GB of free disk is sufficient; the attack builds dense co-occurrence matrices over the ∼30k-document corpora, which is the main memory driver. No GPU, TEE, or FPGA is needed: the `sentence-transformers` embedding model runs on CPU. A CUDA-capable GPU, if present, only speeds up the embedding step.

A.2.4 Software dependencies

Linux (e.g., Ubuntu 22.04/24.04) or macOS, and Python 3.9 or newer (developed and tested on Python 3.13). All Python packages are listed in `requirements.txt` (numpy, scipy, nltk, tqdm, sentence-transformers, torch, openai, python-dotenv, jieba). Three NLTK data packages (`stopwords`, `punkt`, `punkt_tab`) are downloaded once during set-up. Experiments that exercise the LLM-synthesis path additionally require an OpenAI-compatible HTTPS endpoint: the artifact is configured by default for **DeepSeek V4 Flash** (`deepseek-v4-flash`), a low-cost substitute for the Google Gemini 2.5 Flash model used in the paper. We verified that DeepSeek V4 Flash reproduces the paper's headline numbers within ∼1–2 percentage points, so either model is suitable.

A.2.5 Benchmarks

Six public corpora are used: Enron (corporate email), Lucene (Apache developer mailing list), Wikipedia (Simple English), LKML, QEMU, and CC-News. For each dataset the artifact ships a slim keyword/metadata cache (`pickles/<dataset>_5000.pkl`) used by the structural attacks and a gzipped document-body cache (`pickles/<dataset>_bodies.pkl.gz`) used by the LLM-synthesis path. With these caches the experiments are fully self-contained; no raw corpus download is needed.

A.3 Set-up

A.3.1 Installation

From the artifact root, in a clean Python 3.9+ environment:

1. `pip install -r requirements.txt`
2. `python -c "import nltk; nltk.download('stopwords'); nltk.download('punkt'); nltk.download('punkt_tab')"`
3. Before long experiments, pre-cache the embedding model to avoid transient HuggingFace/SSL failures: `python -c "from sentence_transformers import SentenceTransformer; SentenceTransformer('all-mpnet-base-v2')"`
4. `cp .env.example .env`, then set `DEEPSEEK_API_KEY` (the default `DEEPSEEK_BASE_URL` already points to `https://api.deepseek.com/v1`). This step is only needed for experiments that use LLM synthesis (E1–E6 below all do; the basic test can be run without a key by setting `multiplier=0`).

A.3.2 Basic Test

Run the basic functionality test:

```
python -m experiments.compare_val_whisper_leakage
```

The script loads the Enron cache (∼1 min), runs three repetitions at 0.1% leakage, and writes a timestamped log to `results/`. Expected output: a VAL query CRR of ∼17% and a WHISPER CRR of ∼40%, i.e. a strictly positive, roughly 2× gap in every run. Total time ∼7 minutes on a CPU machine.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): At 0.1% leakage, WHISPER improves the correct query recovery rate (CRR) over VAL by roughly 2–2.5× across six datasets (§6.1, Table 3 and Fig. 8). Validated by (E1).

(C2): Recovery scales with leakage, approaching near-complete recovery (exceeding or approaching 80%, up to ∼95%) by 1% leakage, and remains non-trivial even below 0.1% (§6.2, Figs. 9 and 10). Validated by (E2).

(C3): Every WHISPER component contributes to the final CRR: semantic embedding propagation dominates the average gain, LLM synthesis is complementary, and global Hungarian matching beats greedy matching (§6.3, Tables 5 and 7). Validated by (E3).

(C4): Integrating the embedding signal as an early soft prior outperforms late fusion and direct propagation (§6.3, Table 6). Validated by (E4).

(C5): Naively injecting LLM-generated documents into prior attacks (VAL, LEAP) does *not* improve them (§6.2, Table 4). Validated by (E5).

(C6): WHISPER remains effective under the padding and volume-hiding defenses evaluated in the paper, retaining a clear advantage over VAL (§6.4, Fig. 12). Validated by (E6).

A.4.2 Experiments

Each experiment below pairs the command to run (*Execution*) with the expected output and its comparison to the paper (*Results*). Every script is run from the artifact root and writes a timestamped log under `results/`. All scripts ship configured for the 0.1% leakage operating point with three repetitions (`random_seed=45`), the setting that reproduces the reference logs bundled in `results/`. Times are wall-clock on a commodity multi-core CPU with no GPU. Because the shipped LLM differs from the paper's (DeepSeek V4 Flash vs. Gemini 2.5 Flash) and each script uses a single seed, WHISPER numbers may differ from the paper by ∼1–5 pp, while VAL (deterministic) matches closely.

(E1): *Multi-dataset comparison* [5 human-min + ∼8 compute-min]. Reproduces C1 on CC-News.
Execution: `python -m experiments.multi_dataset_validation`
Results: `results/multi_dataset_<ts>.txt` reports VAL vs. Whisper query CRR at 0.1%. Expected: VAL ∼16.7%, Whisper ∼41% (paper Table 3). To cover all six datasets, set `datasets=[...]` in the script's `__main__`.

(E2): *Leakage sweep on Enron* [5 human-min + ∼7 compute-min at 0.1%]. Reproduces C2.
Execution: `python -m experiments.compare_val_whisper_leakage`
Results: `results/val_vs_whisper_leakage_<ts>.txt`. Expected at 0.1%: VAL ∼17.7%, Whisper ∼40%. For the full curve, set `leakage_rates=[0.1,...,1.0]` in `__main__` (∼1–2 h); Whisper then approaches ∼95% by 1%. `test_whisper_limits` characterizes the <0.1% regime (Fig. 10).

(E3): *Component and matching ablation on Enron* [5 human-min + ∼13 compute-min]. Reproduces C3.
Execution: `python -m experiments.ablation_study` and `python -m experiments.ablation_matching_algorithm`
Results: `results/ablation_enron_<ts>.txt` reports the five configurations of Table 5. Expected CRR, ours (paper in parentheses): VAL ∼17.7% (16.7); w/o AI ∼25.1% (24.6); w/o Co-Prop ∼26.2% (27.6); w/o LLM ∼40.0% (39.1); full ∼41.5% (41.0). The matching script reproduces Table 7: Hungarian ∼40% vs. greedy ∼30%.

(E4): *Semantic-fusion mechanism on Enron* [5 human-min + ∼7 compute-min]. Reproduces C4.
Execution: `python -m experiments.`

`compare_external_strategies`
Results: `results/external_strategies_<ts>.txt`. Expected CRR, ours (paper in parentheses): direct propagation ∼30% (30.7); late fusion ∼36% (37.3); soft prior ∼40% (40.6); the soft-prior mechanism is best (Table 6).

(E5): *Naive LLM augmentation of prior attacks* [3 human-min + ∼3 compute-min]. Reproduces C5.
Execution: `python -m experiments.compare_traditional_llm_baselines`
Results: `results/traditional_llm_<ts>.txt`. Expected: each `+LLMdocs` variant is no better than its baseline (CRR change ≤ 0), confirming that naive LLM injection does not help VAL or LEAP (Table 4).

(E6): *Defense sweep on Enron* [5 human-min + ∼22 compute-min]. Reproduces C6.
Execution: `python -m experiments.defend`
Results: `results/defense_<ts>.txt` reports VAL vs. Whisper CRR under no defense, padding (n=500), and volume hiding. Expected (VAL→Whisper): no defense ∼17.9%→41.4%; padding ∼10.4%→20.2%; volume hiding ∼13.6%→33.4%. Padding degrades both attacks the most, yet Whisper keeps a clear (∼2×) advantage over VAL under both (Fig. 12).

Two further scripts produce the implementation-section sensitivity figures: `compare_whisper_multipliers` (Fig. 6) and `compare_blend_weights` (Fig. 7), confirming the defaults $\mu = 1$ and $\alpha = 0.3$.

A.5 Notes on Reusability

Adding a new corpus only requires writing a parser in `utils/email_extraction.py` and registering its path in `MAILDIRS` in `experiments/common.py`; the rest of the pipeline picks it up automatically and caches it on first use. New attacks that follow the constructor and `attack()` interface of the reference attacks in `attacks/` drop into the experiment runners in `experiments/` without changes. The LLM-synthesis path uses a standard OpenAI-compatible client, so any compatible model can be substituted by editing `model_name` and the `DEEPSEEK_*` variables (e.g., to reproduce with the paper's original Gemini 2.5 Flash). Key knobs are exposed in each script's `__main__`: `leakage_rates`, `runs`, `random_seed`, the synthesis `multiplier`, and the embedding `external_blend_weight`; quality-aware vs. uniform known-subset sampling can be toggled via the helpers in `experiments/common.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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: SONIC: Concurrent Oblivious RAM & Data Structures for Low-Latency and High-Throughput

Nihal Talur
UC Santa Cruz

Ioannis Demertzis
UC Santa Cruz

A Artifact Appendix

A.1 Abstract

This artifact contains the SONIC implementation, GraphMap baseline, O2TH and PMCHAIN, and EnigMap/GraphOS baseline artifacts. It supports SGX builds, validation tests, and representative throughput anchors for the paper's single-node evaluation claims.

A.2 Description & Requirements

- `sonic/`: SONIC, GraphMap, O2TH, PMCHAIN.
- `baselines/sgx-baseline-container/`: SGX baseline image.
- `baselines/enigmap/`: EnigMap baseline.
- `baselines/graphos/`: GraphOS baseline.

A.2.1 Security, privacy, and ethical concerns

The artifact uses synthetic workloads and no human-subject or evaluator data. It runs benchmark programs in ephemeral Docker containers with SGX device passthrough. It does not modify host data.

A.2.2 How to access

https://doi.org/10.5281/zenodo.20646912

A.2.3 Hardware dependencies

- x86_64 Linux host with Intel SGXv2
- Azure `Standard_DC32s_v3` instance, with 32 vCPUs, 256 GiB RAM, 192 GiB EPC
- `/dev/sgx_enclave` and `/dev/sgx_provision` (SGX driver is upstreamed since Linux 5.11)

A.2.4 Software dependencies

Docker on x86_64 Ubuntu 22.04. The artifact includes Dockerfiles for the build toolchain.

Host sanity checks:

```
ls -l /dev/sgx_enclave /dev/sgx_provision
docker version
uname -a
docker run --rm \
  --device /dev/sgx_enclave \
  --device /dev/sgx_provision \
  ubuntu:22.04 true
```

Expected: SGX device files exist, Docker functions.

A.2.5 Benchmarks

No external dataset is required as ORAM/OMAP workloads are synthetic. The SONIC demo CLI uses implementation names: `pathoram` is the GraphMap baseline, `zingoram` is SONIC, and `zingoram-disjoint` is disjoint-window SONIC.

A.3 Set-up

A.3.1 Installation

```
# Host shell:
tar -xzf sonic-usenix26-artifacts-20260705.tar.gz
cd sonic-usenix26-artifacts
export HOST_ARTIFACT_ROOT=$PWD
cd "$HOST_ARTIFACT_ROOT/sonic"

docker build -t localhost/sonic2-sgx \
  -f docker/sonic2_sgx.docker .

docker run --privileged \
  --device /dev/sgx_enclave:/dev/sgx_enclave \
  --device /dev/sgx_provision:/dev/sgx_provision \
  --rm -it \
  -e ARTIFACT_ROOT=/artifact \
  -v "$HOST_ARTIFACT_ROOT":/artifact \
  -w /artifact/sonic \
  localhost/sonic2-sgx bash

# SONIC container:
CC=clang CXX=clang++ cmake \
  -G Ninja \
  -B build-sgx-clang \
  -DCMAKE_BUILD_TYPE=Release \
  -DSONIC_PLATFORM=sgx \
```

```
  -DSONIC_DEMO_SGX_MODE=HW \
  -DSONIC_BUILD_APPLICATIONS=ON \
  -DSONIC_BUILD_DISTRIBUTED=ON
cmake --build build-sgx-clang --parallel
BIN="$ARTIFACT_ROOT/sonic/build-sgx-clang/bin"
BIN="$BIN/sonic_demo_sgx"
export BIN

cd
"$HOST_ARTIFACT_ROOT/baselines/sgx-baseline-container"
docker build -t sonic-baseline-sgx .
```

A.3.2 Basic Test

```
# SONIC container:
cd "$ARTIFACT_ROOT/sonic"
# GraphMap
$BIN pathoram validate \
  -N 1024 -B 64 -Z 4 -m 8

# SONIC
$BIN zingoram validate \
  -N 4096 -B 64 -Z 16 -S 16 \
  --routing-depth 3 --evict-batch 2 \
  -T 4 -m 64

# SONIC, disjoint
$BIN zingoram-disjoint validate \
  -N 32768 -B 64 -Z 14 -S 16 \
  --routing-depth 5 --evict-batch 4 \
  -T 4 --disjoint-epoch 17408 -m 128
```

Expected:

```
pathoram validate ok
zingoram validate ok
zingoram-disjoint validate ok
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Artifact builds and validates SONIC and GraphMap. See E1.

(C2): GraphMap, EnigMap, and GraphOS reproduce the Figure 6 baseline results. See E2.

(C3): SONIC reproduces the Figures 7–8 standard ORAM throughput/latency results. See E3.

(C4): SONIC reproduces the Figure 9 thread-scaling trend. See E4.

(C5): SONIC reproduces the Figure 10 deferred-eviction results. See E5.

(C6): O2TH and PMCHAIN benchmarks support the Figure 11 subORAM evaluation story. See E6.

A.4.2 Experiments

For a standard and representative SGX hardware environment, we use the Azure `Standard_DC32s_v3` instance type. The full benchmark matrix in the paper takes $\approx$ 4 VM-days to run for all single-server experiments. The multi-server experiment additionally requires significant effort for provisioning on multiple machines and high ($\approx$ \$800) costs.

Therefore, for artifact evaluation, we recommend running **representative anchors** to corroborate all claims: exact plotted parameter configurations with bounded access counts. This keeps evaluation to a few hours while checking claim-bearing endpoints and scaling trends. The package contains `experiment.py`; run the anchor with `-anchor`. The full run can be started with `-full`.

Expected values are paper-figure op/s. We have observed minor ($\approx$ 5-10%) performance variations across instances on Azure, which falls within expected tolerances; the relative performance story is consistently reproducible.

(E1): Functional validation. [20m human; 1h compute.] Run Basic Test; expect three `validate ok` lines.

(E2): Fig. 6 baselines at $N = 2^{20}$. [15m human; 45m compute.]

```
# Host shell:
cd "$HOST_ARTIFACT_ROOT"
python3 experiment.py baselines --anchor
python3 experiment.py baselines --full
--commands
```

Expected op/s (B=64/256): GraphMap 21639.463/12367.066; EnigMap 13802.878/5098.318; GraphOS 2250.324/1615.670.

(E3): Figures 7–8 SONIC throughput/latency anchors. [5m human; 1h compute.]

```
# Host shell:
cd "$HOST_ARTIFACT_ROOT"
python3 experiment.py scale-n --anchor
python3 experiment.py scale-n --full --commands
```

Expected op/s: Ser/Par8/Par16 = 54209.480/193409.508/254681.666.

(E4): Figure 9 thread-scaling anchors. [5m human; 1h compute.]

```
# Host shell:
cd "$HOST_ARTIFACT_ROOT"
python3 experiment.py scale-t --anchor
python3 experiment.py scale-t --full --commands
```

Expected op/s for T1/T8/T16 = 108184.811/361251.568/450624.877 at $N = 2^{20}$ and 69093.680/240167.842/340705.848 at $N = 2^{24}$.

(E5): Figure 10 deferred-eviction anchors. [5m human; 1h compute.]

```
# Host shell:
cd "$HOST_ARTIFACT_ROOT"
python3 experiment.py disjoint-evict --anchor
python3 experiment.py disjoint-evict --full
--commands
```

Expected op/s: Window/Online = 474060.014/901892.813, with OnlineT16 $>$ WindowT16 $>$ Par16.

(E6): Single-node subORAM benchmarks. [15m human; 1h compute.]

```
# Host shell:
cd "$HOST_ARTIFACT_ROOT"
python3 experiment.py suboram --anchor
python3 experiment.py suboram --full --commands
```

At 512 MiB, E6 tests $K = 90432$ for PMCHAIN and $K = 65536$ for O2TH. Both should have batch latency at most 1s, validating support for at least 90432 and 65536 requests/s, respectively. E6 does not itself directly produce Figure 12; Figure 12 is the separate multi-server experiment requiring nine servers, with one load balancer and eight subORAMs.

A.4.3 Comparing CSVs to the paper

Figure	CSV comparison
6, baseline throughput	`baselines`: compare `throughput_ops_per_sec` by `impl`, `B`, and `N_exp`
7, SONIC throughput	`scale-n`: compare `throughput_ops_per_sec` by `impl`, `B`, and `N_exp`
8, access latency	`scale-n` + `disjoint-evict` (`WindowT16`): compare `mean_access_latency_us`
9, thread scaling	`scale-t`: `T` in `impl` is x; `throughput_ops_per_sec` is y
10, deferred eviction	`disjoint-evict`: compare `throughput_ops_per_sec` by `impl`, `B`, and `N_exp`
11, isolated subORAM	`suboram`: x is `N*B`; y is `K` when `meets_1s_target=true`
12, 1+8-server deployment	Multi-server 1 LB + 8 SO deployment
13, modeled AVL-OMAP	`baselines` + `scale-n`: use ORAM measurements, dividing throughput by AVL-OMAP ORAM access count from `2*h(N)`, where `h(N)` is worst case AVL tree height

A.5 Notes on Reusability

The artifact was carefully designed to be modular and suitable for use as a composable set of libraries for oblivious computation in TEEs. Dependent applications can link the layers and abstractions that they need. The code is also portable and easily retargetable across platforms and architectures.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: DDR-SSE: Duplicated Retrieval of Documents for System-wide Secure Searchable Symmetric Encryption

Zichen Gui
University of Georgia

Simon-Philipp Merz
ETH Zürich

Kenneth G. Paterson
ETH Zürich

Sikhar Patranabis
IBM Research India

A Artifact Appendix

A.1 Abstract

DDR-SSE is a system-wide secure searchable symmetric encryption scheme. This artifact provides a roadmap for the evaluation of DDR-SSE in terms of its efficiency and resilience to leakage-abuse attacks. All results shown in Section 4 and 5 of the paper can be reproduced with the artifact.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We use the Enron email corpus [1] to evaluate DDR-SSE as it is the only public dataset of this kind and it has been used in previous works extensively. The corpus contains emails from real people. Although it has already been sanitized, we advise consumers of the corpus not look at the individual emails directly. Our artifact do not rely on the actual content of the emails, but rather, the statistical properties of them (e.g., the number of keywords in each email).

A.2.2 How to access

The artifact can be accessed via https://doi.org/10.5281/zenodo.17965794.

A.2.3 Hardware dependencies

The artifact can be run on a normal computer. The hardware requirements are:

- CPU: Any modern CPU.
- RAM: 64 GB.
- Storage: 10 GB.

We recommend running the cryptanalysis experiments in Section 4 of our paper on a computer cluster, as these experiments can be parallelized.

[1] https://www.cs.cmu.edu/~enron/

A.2.4 Software dependencies

Our artifact requires Java (JDK 17 and above) and Python (3.8 and above). These can be downloaded and installed from https://www.oracle.com/java/technologies/downloads/ and https://www.python.org/downloads/ respectively.

A.2.5 Benchmarks

The benchmarks and cryptanalysis experiments are run using the Enron email corpus. The dataset can be downloaded from https://www.cs.cmu.edu/~enron/.

A.3 Set-up

A.3.1 Installation

The experiments require a few non-standard Python modules: nltk, numpy, scipy, sklearn, pandas, matplotlib, tqdm, packaging, pytz. These can be installed using pip (https://pypi.org/project/pip/). These modules are listed in requirements.txt, so one can simply run pip install -r requirements.txt to install all dependencies at once.

A.3.2 Basic Test

We do not provide a dedicated basic test of the artifact as it consists of several independent components. Instead, one can check if the components are functioning by running them individually using small parameters. We give more detail in Appendix A.4.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): DDR-SSE is practically efficient. This is proven by the experiment (E1 and E2) whose results are reported in Section 5 of our paper.

(C2): DDR-SSE is resilient to the state-of-the-art leakage-abuse attacks. This is demonstrated by the state-of-the-art leakage-abuse attacks (E5, E6, and E7) and our own attack (E3 and E4).

A.4.2 Experiments

The step-by-step instructions can also be found in README.md of the artifact. For the tasks where the implementation is parallelized (e.g., by using multiprocessing), we report the raw compute time without any parallelization, and the default amount of parallelization. For example, 20 compute-hour (20) means the computation takes 20 hours in total using a single process. With the default amount of parallelization, it should take about an hour.

(E1): [Download and parse the Enron emails] [5 human-minutes + 0.3 compute-hour + 6GB disk]: this step prepares the inputs for Experiments E2 and E3.
Preparation: Download the Enron email corpus. Unzip it into ./emails_raw/.
Execution: Parse the emails by running ./email_parser/email_parser.py. Use the flag --ndoc 1000 to generate small outputs for the basic test. Use flag --ndoc 400000 and --padding p for $p = 4096, 8192$, and 16384 for the full runs.
Results: The files produced are used in the subsequent steps (E2, E3, and E4).

(E2): [Benchmark DDR-SSE] [5 human-minutes + 1 compute-hour]:
Preparation: Navigate to ./DDR-SSE/. Compile the Java code.
Execution: Run Scheme.DDR_benchmark. The program takes three arguments: the number of documents ($1,000$ for the test run, $400,000$ for the full runs), the amount of padding on the encrypted documents ($4,096$ for the test run, $4096, 8192, 16384$ for the full runs), and the bucket size (400 for all runs). Use the Java runtime flag -Xmx55G to give the JVM 55GB memory.
Results: The full runs produce benchmark results in ./DDR-SSE/. Move them to ./benchmark_plots/. Run the plot script in ./benchmark_plots/ to produce Figure 4 in the paper. The other relevant performance statistics are displayed by the script.

(E3): [Leakage extraction] [5 human-minutes + 1 compute-hour + 1GB disk]:
Preparation: Navigate to ./DDR-SSE/. Compile the Java code.
Execution: Run Scheme.DDR_leakage_extraction. The program takes three arguments: the number of documents ($1,000$ for the test run, $400,000$ for the full runs), the amount of padding on the encrypted documents ($4,096$ for all runs), and the bucket size (200 for the test run, $50, 100, 200, 400$ for the full runs). Use the Java runtime flag -Xmx55G to give the JVM 55 GB memory.
Results: The program produces files describing the leakage of DDR-SSE in ./leakage/. Move the files to the folder ./leakage/exact/. These files are used in Experiment E4.

(E4): [Our leakage-abuse attack] [5 human-minutes + 18 compute-days (24) + 1GB disk]:
Preparation: Navigate to ./attack/.
Execution: Run attack_exact_batch.py. For the test run, use argument --nd 1000 --nitr 1000 --nrun 2. For the full runs, use arguments --nbkt b --pct p for $b = 50, 100, 200, 400$ and $p = 100, 95, 90, 85, 80, 75$. We recommend running these jobs on a computer cluster as there are a lot of them.
Alternatively, run script_mp.py on an ordinary computer. The default number of processes used in the script is 20.
Results: The attacks produce reports in ./attack_results/. Run the script attack_results_plot_exact.py in the folder to reproduce Figure 2 in our paper.

(E5): [The SAP attack against DDR-SSE] [5 human-minutes + 20 compute-hours (20)]:
Preparation: Navigate to ./USENIX21-sap-code-master/.
Execution:
1. Run add_experiments_to_manager.py.
2. Run manager_df.py, type w and press return.
3. Run run_pending_experiments.py (adjust the number of cores used for the jobs on line 30 of the Python script if needed).
4. Run manager_df.py. Type eat and press return to consume the attack results.

Results: Run plot5_vs_CLRZ_DDR-SSE.py to reproduce Figure 3(a) in our paper.

(E6): [The IHOP attack against DDR-SSE] [5 human-minutes + 60 compute-hours (20)]:
Preparation: Navigate to ./USENIX22-ihop-code-master/.
Execution:
1. Run add_to_manager.py.
2. Run manager.py, type w and press return.
3. Run run_from_manager.py (adjust the number of cores used for the jobs on line 131 of the Python script if needed).
4. Run manager.py. Type eat and press return to consume the attack results.

Results: Run plot5_vs_CLRZ_DDR-SSE.py to reproduce Figure 3(b) in our paper.

(E7): [The Jigsaw attack against DDR-SSE] [5 human-minutes + 2 compute-hours]:
Preparation: Navigate to ./USENIX24-jigsaw-code-master/.
Execution: Run test_against_CLRZ_DDR-SSE.py.
Results: Run generate_test_against_CLRZ_DDR-SSE.py to reproduce Figure 3(c) in our paper.

A.5 Notes on Reusability

The artifact is split into different modules to make them as reusable as possible.

- The email parser in ./email_parser/ can be adapted to

parse different datasets. It can be used to benchmark other encrypted search schemes.

- The implementation of DDR-SSE in ./DDR-SSE/ supports arbitrary databases (see the email parser for the expected input format). This makes the implementation flexible for (comparative) benchmarks in the future.
- There are two main components in DDR-SSE, namely index retrieval and document retrieval. These components are implemented separately. In the future, if there is a more efficient and/or more secure index retrieval scheme and/or document retrieval scheme, one can substitute that component in DDR-SSE easily.
- Our cryptanalysis attack in ./attack/ can be modified to attack other schemes. Only the score function (line 147 of attack_exact_batch.py) and a few auxiliary functions need to be updated.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: mmCipher: Batching Post-Quantum Public Key Encryption Made Bandwidth-Optimal

Hongxiao Wang
The University of Hong Kong

Ron Steinfeld
Monash University

Markku-Juhani O. Saarinen
Tampere University

Muhammed F. Esgin
Monash University

Siu-Ming Yiu
The University of Hong Kong

A Artifact Appendix

A.1 Abstract

We provide self-contained Python and portable C implementations of the scheme (with some aspects that may need further optimization for production-level deployment). The artifact also contains the code used for generating the comparative benchmarks reported in this work, source code for ZK proof experiments using the LaZer Library, and scripts and tools used for security parameter selection (computation of lattice parameter sets and decryption failure probabilities).

A.2 Description & Requirements

This is a software artifact that has been developed and tested on Ubuntu Linux 24.04 LTE operating system. Running on other UNIX-like operating systems and MacOS should also be straightforward. All external software and library dependencies are standard open-source components.

The artifact contains six directories, each with a separate README file containing running instructions.

Subdirectory	Description
mmCipher-c	Plain C Implementation of mmCipher. Used for benchmarks.
mmCipher-py	Python model for mmCipher, compatible with the C code.
mmCipher-pkzk	ZK proofs of public key validity with LaZer.
refKyber-c	Plain C Kyber ref code for "apples-to-apples" benchmarks.
pr-fail-dec	Computation of decryption/decaps failure probabilities.
param-sage	SageMath lattice parameter selection/exploration scripts.

Note: Only **mmCipher-c** is related to our major claims (of good performance and bandwidth-optimality); other directories serve as "nothing-up-the-sleeve" research artifacts that allow the parameter selection and other aspects of the design to be independently reproduced and validated.

We are including the **mmCipher-pkzk** artifact (for public-key validity proofs) despite it being somewhat difficult to use, as the Lazer library it depends on is in an unstable state. These zero-knowledge proofs are not required during the regular operation of mmCipher and hence do not impact the main claims.

A.2.1 Security, privacy, and ethical concerns

The purpose of the artifact is to reproduce benchmarks and specific mathematical computations related to the mmCipher paper. This software only requires user-level privileges to execute. The software does not interact with other local or remote files (outside its own working directory). Hence, the artifact should not present security, privacy, or ethical concerns.

A.2.2 How to access

The artifact files have been archived as `https://doi.org/10.5281/zenodo.17849532`. The same files may also be accessed as a GitHub repository: `https://github.com/ml-kem/mmcipher-artifact`.

A.2.3 Hardware dependencies

No specific hardware requirements; a standard x86 or ARM workstation/laptop can be used.

A.2.4 Software dependencies

Development was conducted on Linux and MacOS systems; most UNIX-like operating systems should work. The artifact has been tested on Ubuntu 24.04 LTE.

See README files in each subdirectory for a more detailed description:

- **mmCipher-c** (and others): C compiler (gcc or clang) toolchain and build tools (e.g., make).
- **mmCipher-py**: Python 3.11+, with the `pycryptodome` package (`https://pypi.org/project/pycryptodome/`)

- **mmCipher-pkzk**: Uses the lazer library: https://github.com/lazer-crypto/lazer
- **refKyber-c**: C build tools, Python, and bash scripting. To re-create the plots and tables in the paper from performance data (`plot/make_plots.sh`), Python `matplotlib` and `scipy` are required.
- **pr-fail-dec**: GNU Multiprecision (GMP) library (Ubuntu/Debian packages libgmp-dev and libmpfr-dev).
- **param-sage**: Sage is required to run the scripts (tested with SageMath 10.7). https://www.sagemath.org/

A.2.5 Benchmarks

There are no external benchmark requirements.

A.3 Set-up

We will focus on the basic performance and bandwidth benchmarks contained in the `mmCipher-c` subdirectory.

A.3.1 Installation

The C implementation only requires a suitable C compiler toolchain and build tools such as `make`. The exact method of obtaining cycle counts is non-portable; cycle counter access is provided on x86 platforms.

A.3.2 Basic Test

Invoking `make test` performs a test-vector-based functional test on all parameter sets:

```
$ cd mmCipher-c
$ make test
```

A successful run concludes with:

```
ea0510a559aa66e5bbc0859f2544bf7231823ad37606
e892f8865421ba5243bc  testvec.tmp
cmp testvec.tmp testvec.txt
```

The process will stop at compilation errors, and the final file comparison (and the test vector hash) will not match if the implementation does not match with the test vectors.

A.4 Evaluation workflow

The `mmCipher-c` implementation provides facilities for performance testing and functional testing, depending on the TEST variable passed on to the Makefile. The test program created by the Makefile is named `xtest`. To change its behavior, please see the options in the Makefile. For example, to create and run a checksum-generating executable for the 192-bit PKE parameter set:

```
$ make MODE=PKE LEVEL=192 TEST=TESTVEC
$ ./xtest
```

You will have to `make clean` when changing from one parameter set to another, since the parameters are defined as macros.

There are some known-good checksums in `testvec.txt`, in a format also produced by the Python implementation (in the directory `mmCipher-py`). The test vectors consist of lines such as

```
sk[208] chk 8baed64e
```

Where `sk` is the variable (here, secret key), followed by the byte length (here 208), and a 32-bit CRC checksum of the data item.

A.4.1 Major Claims

A multi-message multi-recipient PKE/KEM (mmCipher) enables the batch encryption of multiple messages (as a message vector) for multiple independent recipients in a single operation, significantly reducing costs, particularly bandwidth, compared to the trivial solution of encrypting each message individually using a standard PKE/KEM such as FIPS 203 ML-KEM a.k.a. Kyber.

(C1): The bandwidth requirement of mmCipher is asymptotically optimal: The ciphertext size divided by the number of recipients N approaches a constant as N grows: Figure 7 (and Table 9) in the paper.

(C2): The performance of mmCipher is comparable (or better) to ML-KEM at equivalent security levels: Figure 8 (and Table 8) in the paper.

Note: While we claim sufficient cryptanalytic security (when mmCipher is used appropriately), we do not claim that this implementation is secure against side-channel attacks. Furthermore, some components (especially Gaussian samplers) in the present implementation can be considered as placeholders and must be improved for use in a production environment (which requires further research). More specifically, this code is **not** claimed to be fully constant-time, although an attempt has been made in some segments of the code. However, we expect a "production" implementation to have similar performance characteristics (or better, as the present implementation is in plain C and lacks assembler optimizations).

A.4.2 Experiments

(E1): [60 human-minutes + 15 compute-minutes]:
How to: The basic experiment is to run the mmCipher-C implementation benchmarks; observe the reported ciphertext sizes (Claim C1) and performance metrics (Claim C2).
For performance comparison to ML-KEM, a reference C implementation is reproduced in the `refKyber-c` directory.

Preparation: Optional: Dynamic frequency scaling ("Turbo Boost") and related cache effects can significantly affect reproducibility of performance measurements. A standard practice is to disable it.

Intel-based Linux system: `echo 1 > /sys\ /devices/system/cpu/intel_pstate/no_turbo`

AMD-based Linux system: `echo 0 > /sys\ /devices/system/cpu/cpufreq/boost`

Execution: Move to directory `mmCipher-c` in the artifact. A file `bench.txt` is produced by running `make bench`. This takes a few minutes. See the file `bench-mm.txt` for the data that was used in the paper.
To obtain the ML-KEM (Kyber) comparison data, move to `refKyber-c` directory and run the shell script `./run_bench.sh`. This produces an *another* file named `bench.txt`. See the file `bench-ref.txt` for the data that was used in the paper.

Results: An mmCipher-c benchmark run produces a file `bench.txt` where each non-comment line has: The algorithm parameter set (e.g. mmCipher-PKE-128), the item being measured (e.g. mmSetUp), the number of recipients (e.g. N=1024), and alternatively a bandwidth byte size (e.g. len=) or the timing measurements (cyc= cycle count, sec= wall-clock timing). Note that running on ARM targets will not produce cycle counts; only wall-clock timings will be reported. Secret key and public key sizes can be read from the test vector data file `testvec.txt` (or the generated file `testvec.tmp`).
For comparison purposes, run refKyber-c on the same platform. This produces a `bench.txt` file in a format similar to the produced by mmCipher-c, but also containing parameter data.
One can manually interpret the benchmark results, e.g., by dividing the ciphertext size by the number of recipients (Claim C1) or by comparing the benchmark measurements between mmCipher and Kyber (Claim C2).
Some ad hoc analytic scripts are provided in the `refKyber-c/plot` directory. Running the shell script `make_plots.sh` in that directory recreates files `bytes_relative.pdf` (Figure 7) and `speed_relative.pdf` (Figure 8) of the paper from the data files `../bench-ref.txt` and `../../mmCipher-c/bench-mm.txt`. Different data files can be substituted here if desired.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: "ZipPIR: High-throughput Single-server PIR without Client-side Storage"

Rasoul Akhavan Mahdavi Abdulrahman Diaa Florian Kerschbaum
University of Waterloo
{rasoul.akhavan.mahdavi, abdulrahman.diaa, florian.kerschbaum}@uwaterloo.ca

A Artifact Appendix

A.1 Abstract

Private Information Retrieval (PIR) allows a client to privately access a database without revealing which element is accessed. The limitations of existing PIR protocols conflict with the practical constraints of resource-limited clients and dynamic databases.

To address these challenges, we propose ZipPIR, a high-throughput PIR protocol that compresses LWE ciphertexts into significantly smaller Paillier ciphertexts. ZipPIR leverages the offline phase to obtain this size reduction without incurring the associated computational cost in the online phase. Moreover, under computational assumptions, ZipPIR features an almost silent offline phase, requiring no communication beyond an initial public key, enabling the server to independently generate and update hints during idle times without client interaction. ZipPIR achieves over 2 GB/s of throughput — comparable to state-of-the-art protocols such as SimplePIR — without the need for a large client-stored hint. For PIR over a 1 GB database, ZipPIR has up to 10x higher throughput than existing protocols with no client-side storage, while requiring less than 200 KB of server-side storage per client, significantly enhancing scalability for practical deployments. While prior PIR protocols using Paillier are very inefficient, ZipPIR is the first PIR protocol using Paillier that achieves throughput that is competitive with state-of-the-art PIR protocols.

Our artifact provides the prototype implementation of ZipPIR, used to compare with related work. It also contains all necessary scripts for evaluating related work and comparing the results with ZipPIR.

A.2 Description & Requirements

Our artifact requires a server running Ubuntu 24.04 or higher. The only hardware requirement for this artifact is that the server have support for AVX512. We recommend using a hardware such as an Amazon c5.4xlarge, which has the required support.

A.2.1 Security, privacy, and ethical concerns

Our artifact does not pose any risk for the evaluators.

A.2.2 How to access

Link to artifact on Gitub (recommended for functionality/reproducibility evaluation):

https://github.com/RasoulAM/ZipPIR/tree/a2ffee01accd20c51dbbc69fd2bf9de12f79c0b5

Permanent link on Zenodo:

https://doi.org/10.5281/zenodo.17907224

A.2.3 Hardware dependencies

The only hardware requirement for this artifact is that the server have support for AVX512. We recommend using a hardware such as an Amazon c5.4xlarge, which has the required support. Our evaluation does not rely on multithreading for the results, but non-critical sections of the code are parallelized to save time. We encourage the evaluators to use a machine with many cores to save time.

A.2.4 Software dependencies

Our artifact requires a server running Ubuntu 24.04 or higher. Other software dependencies can be installed using the provided setup scripts.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

A script is provided that performs the entire setup process. This is mentioned in the README of the artifact and the script if called 'setup.sh'. The specific required dependencies are also listed in the README of the artifact.

A.3.2 Basic Test

The script 'run_test.sh' runs the protocol for test parameters. This test should write results to 'test_results.json'.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *ZipPIR is faster than all related work listed in Table 4.*
(C2): *ZipPIR is faster than all relaed work listed in Table 5.*

A.4.2 Experiments

The entire process for reproducing the experimental evaluation is described in the 'benchmarks' directory of the artifact.

(E1): *[Table 4] [Several hours for partial results, over 20 hours of computation for full results]: This experiments compares ZipPIR with other state-of-the-art PIR protocols as a protocol without client-side storage.*
Preparation: *The steps required to install all dependencies and build the artifact are described in the main README of the artifact. (run 'setup.sh' in the root directory of the artifact)*
Execution: *The steps required to run the experiment is described in the README in the benchmarks directory of the artifact. (run 'run-table-4.sh' and 'run-exp.sh' in the 'benchmarks' directory of the artifact)*
Results: *The steps required to interpret the results are described in the README in the benchmarks directory of the artifact. (run 'python3 generate_table4.py' in the 'benchmarks' directory of the artifact). This script prints the results in the 'benchmarks/results' directory in the form of a csv file.*

(E2): *[Table 5] [Less than 30 minutes]: This experiments compares ZipPIR with other state-of-the-art PIR protocols that fall within the category of stateful PIR.*
Preparation: *The steps required to install all dependencies and build the artifact are described in the main README of the artifact. (run 'setup.sh' in the root directory of the artifact).*
Execution: *The steps required to run the experiment is described in the README in the benchmarks directory of the artifact. (run 'run-table-5.sh' in the 'benchmarks' directory of the artifact). This script prints the results in the 'benchmarks/results' directory in the form of a csv file. This only runs ZipPIR, and not related work, since there is no source code available for related work. For that, we refer the reader to the paper.*
Results: *The steps required to interpret the results are described in the README in the benchmarks directory of the artifact. (run 'python3 generate_table5.py' in the 'benchmarks' directory of the artifact). This script prints the results in the 'benchmarks/results' directory in the form of a csv file. This only runs ZipPIR, and not related work, since there is no source code available for related work. For that, we refer the reader to 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: SHADOWFAX: Hybrid Security and Deniability for AKEMs

Phillip Gajland
IBM Research Europe – Zurich

Vincent Hwang
MPI-SP
Radboud University

Jonas Janneck
Ruhr University Bochum

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper **SHADOWFAX: Hybrid Security and Deniability for AKEMs** for hybrid deniable AKEMs. We provide several portable implementations of the hybrid AKEM SHADOWFAX. When instantiated with standardised components (ML–KEM and FALCON), SHADOWFAX yields ciphertexts of 1,728 bytes and public keys of 2,036 bytes, with encapsulation and decapsulation costs of 1.8M and 0.7M cycles on an Apple M1 Pro.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact simply runs the instantiations of cryptographic primitives (non-destructive). See our paper for ethical concerns.

A.2.2 How to access

This artifact is available from the following two sources:

- Zenodo:
  - Available: https://doi.org/10.5281/zenodo.17975204.
  - Final version: https://doi.org/10.5281/zenodo.18939418.
- Github: https://github.com/vincentvbh/shadowfax.

A.2.3 Hardware dependencies

We do not require any particular hardware for the functionality of the artifact. As for the reproducibility of the performance, numbers in our paper were reported on an Apple M1 Pro.

A.2.4 Software dependencies

For the software, we require the following for functionality:

- `gcc`
- `make`
- `bash`
- `cc`

We benchmark the performance of our instantiations with the following software.

- `gcc (Homebrew GCC 13.3.0) 13.3.0`
- `GNU Make 3.81`
- `GNU bash, version 3.2.57(1)-release (arm64-apple-darwin23)`

`cc` is the same as `gcc`.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

For functionality, install the required software with package managers. For reproducing the performance and the same hardware, make sure the GCC version is the same.

A.3.2 Basic Test

Run the commands in Section A.4.2.

A.4 Evaluation workflow

A.4.1 Major Claims

Our major claims are the performance results in Table 1.

Table 1: Cycle counts (in thousands) of different authenticated key encapsulation mechanisms AKEM and ring signature schemes RSig run on a Firestorm core of an Apple M1 Pro running at 3GHz.

RSig	Unit	Gen	Sgn	Ver
Raptor [2, 3]	kcc	71 420	7 980	505
	ms	23.81	2.66	0.17
GANDALF [FALCON, FALCON]* [1]	kcc	-	-	-
	ms	5.20	0.49	0.02
GANDALF [FALCON, FALCON] (this work)	kcc	12 298	832	97
	ms	4.10	0.28	0.03
GANDALF [ANTRAG, MITAKA]	kcc	12 965	1 021	97
	ms	4.32	0.34	0.03
AKEM	**Unit**	**Gen**	**Enc**	**Dec**
DH-AKEM [X25519]	kcc	227	679	457
	ms	0.08	0.23	0.15
PQ-AKEM [BAT, GANDALF [ANTRAG, MITAKA]]	kcc	24 608	1 150	246
	ms	8.20	0.38	0.08
PQ-AKEM [ML–KEM, GANDALF [FALCON, FALCON]]	kcc	13 130	1 198	225
	ms	4.38	0.40	0.07
SHADOWFAX [X25519, BAT, GANDALF [ANTRAG, MITAKA]]	kcc	24 739	1 837	692
	ms	8.25	0.61	0.23
SHADOWFAX [X25519, ML–KEM, GANDALF [FALCON, FALCON]]	kcc	12 785	1 681	659
	ms	4.26	0.56	0.22

* Our benchmark. In [1], the authors accessed the timing counters through the standard C library on our platform. We benchmark their implementation on our platform. For other implementations, we access the cycle counters through the macOS API with a fallback to assembly for cycle counters on other operating systems. As mentioned before, the implementation of [1] leads to slightly larger signatures (1 288 bytes).

A.4.2 Experiments

For the correctness of the implementations, run the following command:

```
bash ./test_everything.sh
```

The script will build and test the correctness of the implementations. The log will be written to `./log/test_log.txt`.

For benchmarking the implementations, run the following command:

```
bash ./bench_everything.sh
```

The script will build and benchmark the implementations. Root access is required on macOS. The results are written to `./log/bench_log.txt` and converted into LaTeXcommands in `./latex/bench_latex.tex`. There are not much human-time involved. As for the compute-time, it requies few minutes or so on a standard personal computer.

A.5 Notes on Reusability

Our artifact is about portable implementations. For functionality, we also tested in the following environment:

- Apple M1 Pro, Sonoma 14.6.1 (same hardware).
  - `Apple clang version 15.0.0 (clang-1500.3.9.4)`.
  - Same `bash` and `make`.
- Dell Inc. XPS 9320, Ubuntu 22.04.5 LTS.
  - `gcc (Ubuntu 11.4.0-1ubuntu1 22.04) 11.4.0`.
  - `GNU Make 4.3`.
  - `GNU bash, version 5.1.16(1)-release (x86_64-pc-linux-gnu`.
- Dell Inc. XPS 9320, Ubuntu 22.04.5 LTS (same hardware as above).
  - `Ubuntu clang version 14.0.0-1ubuntu1.1`.
  - Same `bash` and `make` as above.
- MacBook Pro 2020 (Intel(R) Core(TM) i7-1068NG7 CPU @ 2.30GHz).
  - `Apple clang version 12.0.0 (clang-1200.0.32.28)`.
  - `GNU Make 3.81`.
  - `GNU bash, version 5.2.37(1)-release (x86_64-apple-darwin22.6.0)`

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/usenixsec2026/.

References

[1] Shuichi Katsumata, Guilhem Niot, Ida Tucker, and Thom Wiggers. Comprehensive deniability analysis of signal handshake protocols: X3DH, PQXDH to fully post-quantum with deniable ring signatures. Cryptology ePrint Archive, Paper 2025/1090, 2025. To appear in USENIX Security 2025: 34th USENIX Security Symposium.

[2] Xingye Lu, Man Ho Au, and Zhenfei Zhang. Raptor: A practical lattice-based (linkable) ring signature. pages 110–130, 2019.

[3] Zhenfei Zhang. Raptor. https://github.com/zhenfeizhang/raptor, 2020.



USENIX Security '26 Artifact Appendix: HIJACKKV: A New Threat in Position-Independent KV Cache Reuse

Yichi Zhang[1], Zhiqi Wang[1], Huan Zhang[2], Yuchen Yang[1]
[1]The Pennsylvania State University, [2]University of Illinois Urbana-Champaign
{yichi.zhang, zhiqi.wang, yuchen.yang}@psu.edu, huan@huan-zhang.com

A Artifact Appendix

A.1 Abstract

This artifact provides the reference implementation of HIJACKKV, a KV cache hijacking attack against LLM serving systems that apply position-independent KV cache reuse across tenants. Given a benign document context, HIJACKKV uses a GCG-style discrete optimization to craft a short adversarial token prefix whose cached keys/values, once reused with the victim context under position-independent cache reuse, steer the model toward an attacker-chosen answer.

The artifact contains four main components: (i) the attack CLI `src/run_attack.py`, which optimizes adversarial prefixes and measures attack success; (ii) the re-evaluation CLI `src/run_evaluate.py`, which replays a saved attack under the four implemented cache-recomputation methods (`vanilla`, `random`, `epic`, and `cacheblend`); (iii) four 200-sample QA datasets (HotpotQA, SQuAD, MedQA, and PubMedQA); and (iv) an end-to-end demo script. The artifact reproduces the paper's core attack-success measurements (T-ASR and U-ASR), the chunk-size/cache-ratio/recomputation-ratio robustness sweeps, and the cross-method re-evaluation workflow. This appendix also documents the model-access alternatives, resource budget, and claim-to-command mapping clarified during artifact evaluation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact implements an *offensive* attack on LLM serving and is intended for defensive security research. Reviewers should be aware of the following:

- **No destructive operations.** The code performs no privileged, destructive, or system-modifying actions. It reads the bundled datasets, downloads model weights when requested, and writes results under `.data/results/`. No data leaves the local machine and no network service is contacted other than the HuggingFace model/dataset hub used for optional model downloads.
- **Dual-use.** The output is a set of adversarial token prefixes. The supplied workflow evaluates them against locally hosted models in a controlled environment; it does not target a production service.
- **Data.** The bundled datasets are derived from public QA benchmarks and contain no sensitive information.
- **Responsible disclosure.** The vulnerability class (unsafe KV reuse across requests) and corresponding mitigations are discussed in the paper. The artifact is released to help defenders reproduce and test the threat.

A.2.2 How to access

The versioned artifact is available from Zenodo the concept DOI https://doi.org/10.5281/zenodo.20403785 resolves to the permanent record for the artifact family. The source repository is also available at https://github.com/YichiCS/KV-Cache-Hijack.

A.2.3 Hardware dependencies

A CUDA-capable NVIDIA GPU is required. The reference experiments were run on Ubuntu 22.04.5 LTS with two AMD EPYC 9334 32-Core CPUs and four NVIDIA RTX PRO 6000 Blackwell Workstation Edition GPUs (97,887 MiB each). The attack runs on a single GPU; multiple GPUs launch one worker per listed device and primarily reduce wall-clock time. A GPU with roughly 48 GB of memory is a practical starting point for an 8B model, although the reference attack can peak at approximately 48–60 GiB depending on the settings.

On the reference GPU, GCG optimization takes approximately 15 minutes per sample with the default settings. With work distributed over four GPUs, 50 samples take about 3.1 hours, excluding model loading, process scheduling, and later evaluation. These are planning estimates and the detailed settings can change the runtime.

A.2.4 Software dependencies

- Linux with an NVIDIA driver supporting CUDA and PyTorch 2.10.
- Python `>=3.12,<3.13` and the `uv` package manager (https://astral.sh/uv).
- Dependencies in `pyproject.toml` / `uv.lock`.
- A HuggingFace account with access to the gated `meta-llama/Llama-3.1-8B-Instruct` checkpoint, or a local checkpoint supplied through `--model`.

An FP16 8B checkpoint occupies approximately 16 GB. The four bundled datasets occupy less than 1 MiB in total; reserve additional space for the Python environment, model cache, and repeated experiment outputs.

A.2.5 Models and cache backends

The principal RQ1 models are

- `meta-llama/Llama-3.1-8B-Instruct`
- `Qwen/Qwen3-8B`
- `mistralai/Ministral-8B-Instruct-2410`

The cross-model RQ4 targets are Qwen3-4B, Qwen3-8B, Qwen3-14B, Llama-3.2-1B, Llama-3.2-3B, and Llama-3.3-70B. The model identifier can be a HuggingFace name or an absolute path to an already downloaded checkpoint.

The `random` and `cacheblend` implementations use model-internal attention code and, in the released implementation, support Llama- and Qwen3-style backends. The `vanilla` and `epic` paths are the lightweight choices for a first smoke test. A model/backend combination should therefore be checked with the basic test before launching a full sweep.

A.2.6 Benchmarks

Four ready-to-use 200-sample datasets are bundled under `.data/datasets/`:

- `hotpotqa_200.json`: HotpotQA;
- `squad_200.json`: SQuAD (v1.1/v2.0);
- `medqa_200.json`: MedQA; and
- `pubmedqa_200.json`: PubMedQA.

Each record contains `id`, `context`, `question`, `answer`, and an attacker-chosen `target`. The original public sources are HotpotQA (https://hotpotqa.github.io/), SQuAD (https://rajpurkar.github.io/SQuAD-explorer/), MedQA (https://github.com/jind11/MedQA), and PubMedQA (https://pubmedqa.github.io/).

A.3 Set-up

A.3.1 Installation

Run the following commands from the extracted artifact root:

1. Install `uv` if it is not already available:
   ```
   curl -LsSf https://astral.sh/uv/install.sh | sh
   ```
2. Create the locked environment:
   ```
   uv sync
   ```
3. If the gated Llama checkpoint is downloaded online, install the optional HuggingFace CLI and authenticate with an individually approved account:
   ```
   uv pip install 'huggingface_hub[cli]'
   uv run huggingface-cli login
   ```

A shared author token is neither required nor provided. If a reviewer already has a legitimately obtained checkpoint, the model can be passed directly, for example:

```
MODEL=/path/to/local-llama
--model "$MODEL"
```

A.3.2 Basic Test

The following four-sample test avoids gated-model access and exercises the attack CLI, the vanilla recomputation path, and the result-writing code. On a 48 GB-class GPU, keep `--eval_batch_size 32`; reducing it further changes candidate batching but not the GCG optimization results.

```
uv run python src/run_attack.py \
  --dataset .data/datasets/hotpotqa_200.json \
  --model Qwen/Qwen3-8B \
  --device 0 \
  --max_samples 4 \
  --cache_ratio 0.3 \
  --chunk_size 32 \
  --gcg_recomp_method vanilla \
  --gcg_recomp_ratio 0.1 \
  --eval_recomp_method vanilla \
  --eval_recomp_ratio 0.1 \
  --eval_batch_size 32
```

A successful run prints per-sample progress lines such as `[Worker 0] cuda:0 | 1/4 | id=... | loss=... | step=...` and writes an experiment directory under `.data/results/`. Its `metrics.json` contains a `metric` block with T-ASR/U-ASR fields and the per-sample attack payload.

A.4 Evaluation workflow

A.4.1 Metrics and result interpretation

All commands below are run from the artifact root inside the `uv` environment. The result summary is stored in the `metric` block of `metrics.json` or of a file under `<experiment\_dir>/eval/`. Let r be the ground-truth answer, $\tilde{r}$ the attacker-chosen target, x the benign output, and y the output under the attack. The reported metrics are:

- **Accuracy (Acc):** $x = r$, measuring benign task utility.
- **U-ASR:** $y \neq x$, measuring whether the attack changes the model output.

- **T-ASR:** $y = \tilde{r}$ and $y \neq x$, measuring targeted manipulation success.

The `-gcg_recomp_method` and `-gcg_recomp_ratio` control recomputation simulated during prefix optimization. The `-eval_recomp_method` and `-eval_recomp_ratio` control the target-system evaluation pass. Keeping these pairs separate makes it possible to test adaptive attacks and mismatched defenses.

A.4.2 Major Claims

The paper's four major claims and their corresponding experiments are:

(C1): HIJACKKV can manipulate model outputs through position-independent KV cache reuse, achieving targeted and untargeted attack success across the evaluated QA datasets, models, and cache methods. This is evaluated by (E2) and corresponds to Table 1.

(C2): The attack remains effective across cache-system parameters, including chunk size, cache ratio, and recomputation ratio. This is evaluated by (E3) and corresponds to Tables 2–4.

(C3): Adversarial prefixes persist when unrelated filler passages are inserted between the cached context and the query. This is evaluated by (E4) and corresponds to Table 5.

(C4): Adversarial prefixes transfer across model identifiers and tokenizers in the intended black-box evaluation setting. This is evaluated by (E5) and corresponds to Table 6.

The paper's *Full* rows are benign no-cache reference measurements and do not report attack metrics. The direct attack/re-evaluation workflow provided here covers the `vanilla`, `random`, `epic`, and `cacheblend` paths.

A.4.3 Experiments

The following experiments provide the claim-to-command mapping. The estimates use the reference four-GPU server described above; smaller test runs can be used to validate the installation before starting a full experiment.

Shared defaults. Unless a command specifies otherwise, use `-cache_ratio 0.3`, `-chunk_size 32`, `-gcg_recomp_ratio 0.1`, and the default GCG budget (250 steps, top-k 512, search width 256). Use `-device 0,1,2,3` on the reference server. For a 48 GB-class L40S, start with `-max_samples 4` and `-eval_batch_size 32` before launching a 200-sample run.

(E1) End-to-end demo [15 human-minutes + about 3.5 compute-hours + 1 GB disk]. Set the dataset and device variables near the top of `scripts/demo.sh` to paths valid on the local machine. If the script contains a reference-machine absolute dataset path, replace it with `.data/datasets/hotpotqa\_200.json`. If the default gated model is not available, pass an open model or a local checkpoint through the equivalent CLI command below. Then run:

```
bash scripts/demo.sh
```

The demo attacks 50 HotpotQA samples with vanilla GCG optimization at recomputation ratio 0.1 and re-evaluates the saved prefixes under `vanilla`, `random`, `epic`, and `cacheblend`. It is a functionality smoke test for the attack, result-writing, and cross-method evaluation paths; the full claim experiments are (E2)–(E5).

(E2) RQ1 / Table 1: attack effectiveness [15 human-minutes + about 12.5 compute-hours per 200-sample run]. For one dataset/model combination, run the following command with `-max_samples 200`. Repeat it for the four dataset files and for the three RQ1 models. To obtain the four cache-method columns, either run the following loop or re-evaluate one saved experiment under each method:

```
MODEL=meta-llama/Llama-3.1-8B-Instruct
DATASET=.data/datasets/hotpotqa_200.json

for M in vanilla random epic cacheblend; do
  uv run python src/run_attack.py \
    --dataset "$DATASET" \
    --model "$MODEL" \
    --device 0,1,2,3 \
    --max_samples 200 \
    --cache_ratio 0.3 \
    --chunk_size 32 \
    --gcg_recomp_method "$M" \
    --gcg_recomp_ratio 0.1 \
    --eval_recomp_method "$M" \
    --eval_recomp_ratio 0.1 \
    --eval_batch_size 32
done
```

The default Llama identifier requires the evaluator's own HuggingFace access or a local checkpoint. The resulting T-ASR/U-ASR metrics support C1.

(E3) RQ2 / Tables 2–4: system-parameter robustness [20 human-minutes + about 12.5 compute-hours per sweep point]. *Chunk size (Table 2).* Run the attack five times with $L_{\text{chunk}} \in \{32, 64, 128, 256, 512\}$, keeping $\delta = 0.3$ and $\rho = 0.1$:

```
MODEL=meta-llama/Llama-3.1-8B-Instruct
DATASET=.data/datasets/hotpotqa_200.json
for C in 32 64 128 256 512; do
  uv run python src/run_attack.py \
    --dataset "$DATASET" \
```

```
    --model "$MODEL" \
    --device 0,1,2,3 \
    --max_samples 200 \
    --chunk_size "$C" \
    --cache_ratio 0.3 \
    --gcg_recomp_method vanilla \
    --gcg_recomp_ratio 0.1 \
    --eval_recomp_method vanilla \
    --eval_recomp_ratio 0.1 \
    --eval_batch_size 32
done
```

For every generated experiment directory, re-evaluate the saved prefixes under all four target methods at $\rho = 0.1$:

```
for M in vanilla random epic cacheblend; do
  uv run python src/run_evaluate.py \
    --dataset <EXPERIMENT_DIR> \
    --model "$MODEL" \
    --device 0 \
    --eval_recomp_ratio 0.1 \
    --eval_recomp_method "$M"
done
```

Cache ratio (Table 3). Repeat the same attack/evaluation workflow with $\delta \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ and `-gcg_recomp_ratio 0.1`:

```
DATASET=.data/datasets/hotpotqa_200.json
for D in 0.1 0.2 0.3 0.4 0.5; do
  uv run python src/run_attack.py \
    --dataset "$DATASET" \
    --model "$MODEL" \
    --device 0,1,2,3 \
    --max_samples 200 \
    --cache_ratio "$D" \
    --chunk_size 32 \
    --gcg_recomp_method vanilla \
    --gcg_recomp_ratio 0.1 \
    --eval_recomp_method vanilla \
    --eval_recomp_ratio 0.1 \
    --eval_batch_size 32
done
```

Then use the preceding `run_evaluate.py` loop for each generated experiment directory. The cache ratio is the proportion of the context occupied by matched chunks; it is distinct from the recomputation ratio.

Recomputation ratio (Table 4). For each method and $\rho \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$, optimize and score with the same method and ratio:

```
DATASET=.data/datasets/hotpotqa_200.json
for M in random epic cacheblend; do
  for R in 0.1 0.2 0.3 0.4 0.5; do
    uv run python src/run_attack.py \
      --dataset "$DATASET" \
      --model "$MODEL" \
      --device 0,1,2,3 \
      --max_samples 200 \
      --cache_ratio 0.3 \
      --chunk_size 32 \
      --gcg_recomp_method "$M" \
      --gcg_recomp_ratio "$R" \
      --eval_recomp_method "$M" \
      --eval_recomp_ratio "$R" \
      --eval_batch_size 32
  done
done
```

The saved experiment directory printed by each run is the input to `run_evaluate.py`. Because each full run is expensive, use the four-sample basic test to validate model/backend compatibility first. The resulting sweep metrics support C2.

(E4) RQ3 / Table 5: multi-turn persistence [30 human-minutes + approximately 1 compute-hour, depending on the target model]. RQ3 uses HotpotQA and keeps the optimized prefix fixed while inserting unrelated passages from other HotpotQA examples between the cached context and the target query. Construct five JSON variants with filler lengths $L_{\text{context}} \in \{0, 256, 512, 1024, 2048\}$ model tokens. Preserve the original `id`, `question`, `answer`, and `target`; insert the filler only at the context/query boundary and preserve each record's `gcg_prefix_ids` from the base attack result. Re-evaluate each variant with `run_evaluate.py` and compare the metrics across filler lengths; these results support C3.

(E5) RQ4 / Table 6: black-box transferability [20 human-minutes + about 12.5 compute-hours for proxy optimization, plus target evaluation]. First optimize the prefixes on the Llama-3.1-8B proxy model:

```
uv run python src/run_attack.py \
  --dataset .data/datasets/hotpotqa_200.json \
  --model meta-llama/Llama-3.1-8B-Instruct \
  --device 0,1,2,3 \
  --max_samples 200 \
  --cache_ratio 0.3 \
  --chunk_size 32 \
  --gcg_recomp_method vanilla \
  --gcg_recomp_ratio 0.1 \
  --eval_recomp_method vanilla \
  --eval_recomp_ratio 0.1 \
  --eval_batch_size 32
```

Let `<EXPERIMENT_DIR>` be the resulting directory. Re-evaluate the same saved prefixes with each target model:

```
for TARGET in \
  Qwen/Qwen3-4B \
  Qwen/Qwen3-8B \
  Qwen/Qwen3-14B \
  meta-llama/Llama-3.2-1B \
  meta-llama/Llama-3.2-3B \
  meta-llama/Llama-3.3-70B; do
  uv run python src/run_evaluate.py \
    --dataset <EXPERIMENT_DIR> \
    --model "$TARGET" \
    --device 0 \
    --eval_recomp_ratio 0.1 \
    --eval_recomp_method vanilla
done
```

Repeat the evaluation loop for the other three recomputation methods. When the proxy and target tokenizers differ, the evaluator records the target-model metrics using its cross-tokenizer path; this is the intended black-box setting. Large target models require a correspondingly large-memory GPU. The resulting cross-model metrics support C4.

A.5 Notes on Reusability

The attack and evaluation interfaces accept any JSON list, or an object with a top-level `result` list, whose records provide `context`, `question`, `answer`, and `target`. The model is selected through `-model`; the cache-system controls are `-chunk_size`, `-cache_ratio`, `-gcg_recomp_*`, and `-eval_recomp_*`. The GCG budget can be varied with `-prefix_chunk_num`, `-num_steps`, `-topk`, `-search_width`, and `-eval_batch_size`.

The attack performs optimization locally and does not require access to a victim cache, victim prompts, or model weights beyond the locally selected surrogate. The resulting prefixes can be saved and re-evaluated under a different model or cache-recomputation method, which supports adaptive-attack, transferability, and defense-oriented studies beyond the paper's default settings.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Low-Cost Hard-Label Adversarial Attack with Theoretical Foundations

Jun Liu[†‡], Leo Yu Zhang[§], Fengpeng Li[†], Isao Echizen[¶‡], Jiantao Zhou[†*]

[†]University of Macau, [‡]National Institute of Informatics, [§]Griffith University, [¶]University of Tokyo
[*]Corresponding author

A Artifact Appendix

A.1 Abstract

This artifact contains the code, scripts, checkpoints, configuration files, and documentation for reproducing the main results of *Low-Cost Hard-Label Adversarial Attack with Theoretical Foundations*. It implements DPAttack, a query-efficient untargeted hard-label black-box adversarial attack with a zero-query frequency-based initialization and a Pattern-Driven Optimization (PDO) module. The artifact supports experiments on CIFAR-10, ImageNet, ObjectNet/CLIP, ImageNet-C, PathMNIST, adversarially trained models, real-world commercial APIs, Blacklight, randomized defenses, and dense prediction tasks. It also includes a lightweight CIFAR-10 Quick Start with Docker support, intended for rapid functional verification before running larger experiments.

This artifact evaluation appendix focuses on the experimental pipeline for the major evaluation settings, including CIFAR-10, ImageNet, ObjectNet/CLIP, adversarially trained models, and the Blacklight defense, to facilitate efficient artifact assessment.

A.2 Description & Requirements

The full artifact provides: (1) source code for the proposed DPAttack method, including fixed-block-size, dynamic-block-size, ℓ_2, Blacklight-aware, and randomized-defense variants; (2) scripts and wrappers for reproducing the main experiments and baselines; (3) analysis utilities for BFS, frequency statistics, gradient-alignment, cosine-similarity evolution, and curvature estimation; (4) included or referenced checkpoints and auxiliary files; and (5) README files describing dataset preparation, expected directory structure, hardware/runtime requirements, Quick Start usage, and commands for reproducing the paper's main tables and figures.

A.2.1 Security, privacy, and ethical concerns

This artifact contains adversarial-attack code and therefore could potentially be misused to study or evade model robustness in real-world systems. However, it does not include destructive functionality, malware, privilege escalation, persistence, or denial-of-service components. The code is released for robustness evaluation and defensive research only. Some experiments target commercial APIs; those require the evaluator's own credentials and should be conducted responsibly, with conservative query budgets and in compliance with the corresponding service policies.

A.2.2 How to access

The artifact is publicly available via the permanent Zenodo DOI https://doi.org/10.5281/zenodo.20322560, as provided in the Open Science section of the paper. It consists of three compressed packages: `DPAttack.tar.gz`, which contains the full artifact, including the code for E1–E3; `CIFAR10_QuickStart.tar.gz`, which provides a self-contained quick-start package with a Dockerfile; and `TestImages.tar.gz`, which includes 500 test images each from ObjectNet and ImageNet.

A.2.3 Hardware dependencies

Our experiments were primarily conducted on a single NVIDIA RTX A6000 GPU with 48GB VRAM. This is the hardware used in our experiments, not a strict minimum requirement.

Table 1: Observed hardware usage for representative settings.

Setting	Observed Usage
CIFAR-10 Quick Start	$<$ 1GB GPU VRAM, $<$ 1GB RAM
ImageNet classifiers	$<$ 2GB GPU VRAM, around 2GB RAM
CLIP-based classification	Up to around 14GB GPU VRAM, around 2GB RAM

For example, in our setup, ConvNeXt-Base used approximately 1.9GB system RAM. Among the full experiments, CLIP-based attacks had the largest GPU memory footprint. Running the full suite of experiments and baselines may take several days, depending on hardware, query budgets, dataset

availability, API rate limits, and whether all baselines are rerun. We therefore recommend starting from the Quick Start and then reproducing representative experiments.

A.2.4 Software dependencies

We use Ubuntu with Conda for environment management, and recommend Python 3.11. Unless otherwise specified in the corresponding sections, the test experiments described below can be executed directly in the environment provided by the Quick Start image. For the complete set of experiments reported in the paper, we provide the full pinned Conda environment file: `tools/py311_20260517.yaml`.

For the Quick Start setup, we additionally provide `Dockerfile.quickstart`, `requirements_quickstart.txt`, and `requirements_quickstart_docker.txt`. Depending on the local Docker configuration, Docker commands may require `sudo`. Some experiments further require external public datasets, pretrained checkpoints, third-party repositories, or commercial API credentials; these requirements are documented in the repository README or explicitly noted in the corresponding experiment sections.

A.2.5 Benchmarks

The artifact evaluation supports experiments on CIFAR-10, ImageNet-1K and ObjectNet/CLIP. We provide small subsets of the public datasets for efficient evaluation, while the full large-scale public datasets and some third-party repositories are not redistributed directly. The following sections and README explain how to obtain these resources and place them in the expected directory structure.

A.3 Set-up

A.3.1 Installation

Quick start.

1. Download the artifact `CIFAR10_QuickStart.tar.gz`.
2. Extract this package:
   ```
   tar -xzf CIFAR10_QuickStart.tar.gz
   cd CIFAR10_QuickStart
   ```
3. Build the Docker image:
   ```
   docker build -f Dockerfile.quickstart \
   -t dpattack-cifar10-quickstart:latest .
   ```

If Docker is not available in your environment, please follow the instructions in `CIFAR10_QuickStart/README_quickstart.md` to install the required dependencies with `pip`.

Full artifact. The following steps 2-3 are optional for artifact evaluation. The Conda environment built in the Quick Start above can be reused for the following evaluation experiments, requiring only a small number of additional package installations when necessary.

1. Extract the full artifact[1] `DPAttack.tar.gz`.
2. Create the Conda environment:
   ```
   conda env create -f tools/py311_20260517.yaml
   conda activate py311
   ```
3. Prepare datasets / checkpoints / third-party repositories according to the README.

A.3.2 Basic Test

The basic functionality test is the CIFAR-10 Quick Start. It is designed to finish in about 5 minutes and serves to verify that the software environment, checkpoint loading, and attack pipeline are functioning correctly.

Docker command:

```
docker run --rm --gpus all dpattack-cifar10- \
quickstart:latest
python OursCifar10DBS.py --victimmodel vgg
```

If the system uses the NVIDIA runtime in CDI mode, use:

```
docker run --rm --runtime=nvidia \
-e NVIDIA_VISIBLE_DEVICES=all \
-e NVIDIA_DRIVER_CAPABILITIES=compute,utility \
dpattack-cifar10-quickstart:latest
python OursCifar10DBS.py --victimmodel vgg
```

Expected result: The script should successfully load the CIFAR-10 victim model VGG-16-BN, attack 200 test samples, and produce summary statistics consistent with those reported in Table 10 of the manuscript, including an Attack Success Rate (ASR) of approximately 30% under a query budget of 50. It reports the ASR and query statistics (Avg.Q and Median.Q). By default, the result log is saved under `CIFAR10_QuickStart/results/` with an automatically generated file name. Users can set `--results_format` to either `json` or `csv`, and use `--results_log` to specify a custom output path.

A.4 Evaluation workflow

A.4.1 Major Claims

This artifact is designed to reproduce the following major claims of the paper:

(C1): *On standard classifiers, especially ImageNet, DPAttack achieves higher ASR and better query efficiency than prior hard-label attacks under realistic query budgets. This is supported by Sec.5.2 and Table 3.*

(C2): *DPAttack generalizes beyond standard ImageNet models to CLIP/ObjectNet. This is supported by Secs.5.3 and Tables 4.*

(C3): *DPAttack remains effective under Blacklight. This is supported by Sec.5.4 and Table 7.*

[1] Please note that `CIFAR10_QuickStart` contain utility files whose names overlap with those in the full `DPAttack` artifact. To avoid ambiguity, please run the corresponding commands within their respective workspaces when invoking these overlapping files.

A.4.2 Experiments

The results reported below are expected to be close to those in the paper. Small numerical differences may still arise from hardware/software differences, such as CUDA/cuDNN versions, dependency versions, and GPU computation details.

(E1): *Representative ImageNet experiments for C1 [20 human-minutes + 9 compute-minutes].*

How to: Reproduce a representative attack experiment on ImageNet dataset against ResNet-50 with a fixed query budget of 50, as reported in Table 3.

Preparation: Activate the Conda environment built by the Dockerfile in the Quick Start. Extract `TestImages.tar.gz` and move the extracted `val` directory to `DPAttack/data/imagenet/`.

Execution: Run the command `python OursDBS.py`.

Results: The expected trend is that `Ours_dyn` achieves nearly the same ASR ($\approx 55\%$) and query efficiency as reported in Table 3. After execution, the key summary metrics, including `ASR`, `Avg.Q`, and `Median.Q`, will be saved under `DPAttack/results/`.

Note: The results for `Ours_opt` and the other baselines in Table 3 can also be reproduced using:

```
python Ours.py
python baselines/ADBA.py
python baselines/attack_imagenet_others.py \
--attack_method rays
```

These additional experiments require approximately 6 compute-minutes, 2 compute-minutes, and 1/2/2/2/2 compute-minutes for HRayS/TtBA/BouT/BouR/HSJA.

(E2): *Representative ObjectNet experiments for C2 [20 human-minutes + 1 compute-hour].*

How to: Reproduce attack experiments on ObjectNet dataset against the CLIP model with a fixed query budget of 50, as reported by Table 4.

Preparation: Activate the Conda environment built by the Dockerfile in the Quick Start. Install the CLIP model dependencies using:

```
pip install open_clip_torch transformers
```

Extract `TestImages.tar.gz` and move the extracted `objectnet_sample` directory to `DPAttack/data/objectnet/`.

Execution: Run the command

```
python OursDBS.py --victimmodel clip \
--datasource objectnet
```

to perform `Ours_dyn` on ObjectNet dataset.

Results: The expected trend on ObjectNet/CLIP is that `Ours_dyn` outperforms ADBA and the other baselines in terms of ASR ($\approx 39.5\%$) and query efficiency under realistic query budgets, consistent with Table 4. After execution, the key summary metrics, including `ASR`, `Avg.Q`, and `Median.Q`, will be saved under `DPAttack/results/`. This experiment also illustrates the most memory-demanding classification setting in the artifact.

Note: The results for `Ours_opt` and the other baselines on ObjectNet/CLIP in Table 4 can also be reproduced using the following scripts:

```
python Ours.py --victimmodel clip \
--datasource objectnet --blocksize 4
python baselines/ADBAObjectNetClip.py
python baselines/attack_objectnet_others.py\
--attack_method rays
```

These additional experiments require approximately 30 compute-minutes, 30 compute-minutes, and 15/30/15/15/15 compute-minutes for HRayS/TtBA/BouT/BouR/HSJA, respectively.

(E3): *Representative Blacklight experiments for C3 [20 human-minutes + 2 compute-hours].*

How to: Reproduce the ImageNet attack experiments against ViT-B-32 under the stateful detection defense Blacklight, with a fixed query budget of 100, as reported in Table 7.

Preparation: Activate the Conda environment built by the Dockerfile in the Quick Start.

Execution: Run `python OursBlacklight.py` to perform `Ours_dyn` against ViT-B-32 on ImageNet.

Results: The expected trend is that `Ours_dyn` remains effective under Blacklight and achieves stronger attack and evasion performance than competing methods, consistent with Table 7. Specifically, DPAttack is expected to achieve a 0% Blacklight detection rate and an ASR of nearly 43%. In this setting, once an input sample is detected by the defense, the attack on that sample is terminated and counted as a failure. After execution, the attack metrics together with Blacklight-related defense statistics, including the numbers of successful and failed detections, the Blacklight detection ratio, and the average and median query indices at which Blacklight first detects an attack, will be saved under `DPAttack/results/`.

Note: The results for `Ours_opt` and the other baselines in Table 7 can also be reproduced using the following scripts:

```
python OursBlacklightOpt.py --blocksize 4
python baselines/ADBA.py --victimmodel vit \
--budget 100 --defense 1
python baselines/attack_imagenet_others.py \
--attack_method rays --defense 1 \
--targetn vit --maxquery 100
```

These additional experiments require approximately 2 compute-hours, 2.5 compute-hours, and 2/5/5/5/2 compute-minutes for HRayS/TtBA/BouT/BouR/HSJA, respectively.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: BUIzz: Finding Policy Enforcement Bugs via Interaction Simulation on the Browser User Interface

Mingi Jung *UNIST*
Donggyu Kim *UNIST*
Mijung Kim *UNIST*
Seongil Wi *UNIST*

A Artifact Appendix

A.1 Abstract

This artifact contains the implementation of BUIzz, including its COLLECTOR, SIMULATOR, and DETECTOR components, as well as the policy-specific test servers used in our evaluation. The artifact also includes a minimal working example (`example/`) that allows evaluators to run the complete detection workflow and reproduce representative policy-enforcement bugs reported in the paper, including a CVE-assigned vulnerability.

Because BUIzz operates by simulating Browser User Interface (BUI) interactions at the operating-system level, it requires an interactive Windows desktop session with a real display and cannot run headless or in Continuous Integration (CI) environments. Beyond reproducing the paper's findings, the artifact can serve as a foundation for future research on browser security testing.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

While running, BUIzz controls the physical mouse and keyboard, and serves local test pages via `hosts`-file redirection with an `mkcert` root CA. We recommend a spare machine and undoing these host changes afterwards. To reproduce already-patched bugs without downgrading an installed browser, use the version-pinned portable Brave in `example/`. All experiments are conducted locally and do not interact with external services or user data.

A.2.2 How to access

Source code with a permanent DOI: https://doi.org/10.5281/zenodo.20422485.

A.2.3 Hardware dependencies

A single x86-64 machine running an interactive Windows desktop session with an attached display is required. No special hardware (GPU/FPGA) is required.

- **Display:** a real, single primary monitor at a standard resolution. The SIMULATOR computes on-screen link coordinates for OS-level input, so it requires a foreground window and cannot run headless.
- **Memory/disk:** the paper used commodity desktops with 8–32 GB RAM (Table 6); 16 GB RAM and ∼50 GB of free disk are comfortable for the functional tests in this artifact.
- **Recommended display:** a single 2560×1440 primary monitor.

A.2.4 Software dependencies

Tested on Windows 11 with Python 3.11.

- **OS:** Windows 10/11 (64-bit).
- **Python:** Python 3.11.x with `playwright` 1.53, `pyautogui` 0.9.54, `pywinauto` 0.6.9, `psutil`, `pywin32`, `webdriver-manager`, and `mysql-connector-python` 9.0.0.
- **Database:** MySQL 8.0.x running on `localhost:3306`. By default, the artifact uses the username `root` and password `1234`. These defaults can be overridden using the `DB_HOST`, `DB_USER`, and `DB_PASSWORD` environment variables. `makeDB.py` automatically creates the required database and tables.
- **Containers:** Docker Desktop with `docker compose` support.
- **TLS:** `mkcert` (installed automatically by `certs.ps1`).
- **Browsers:** Chrome 138, Firefox 139, Edge 136, Opera 118, Brave 1.79, and Whale 4.32 were used in the paper. Since several reported bugs have already been patched in newer releases, the artifact includes a version-pinned portable Brave browser for reproducibility. The minimal example downloads and uses this browser automatically. Note that Chrome must also be installed, because it serves as the baseline browser for all Chromium-based targets (see Section A.4.2).

A.2.5 Benchmarks

- **Datasets:** the test pages reused from prior work (DiffCSP, XSR-Framework) and WPT (Table 2 of the paper),

all included in the repository. No external download is required.

- **Evaluation metric:** pre/post-interaction enforcement inconsistencies, grouped into distinct bugs. See `example/` for the end-to-end reproduction and the expected output.

A.3 Set-up

A.3.1 Installation

1. Install Python 3.11, MySQL 8.0, and Docker Desktop. Ensure that both MySQL and Docker are running before continuing.
2. From the repository root, run `setup.ps1` in a PowerShell session running as Administrator. This script installs the required Python dependencies, downloads the Playwright browser binaries, and configures the test domains in the Windows `hosts` file.
   ```
   PowerShell -ExecutionPolicy Bypass
   -File setup.ps1
   ```
3. Generate the local TLS certificates used by the test servers:
   ```
   PowerShell -ExecutionPolicy Bypass
   -File certs.ps1
   ```
4. Create the database schema required by the artifact:
   ```
   python makeDB.py
   ```

A.3.2 Basic Test

As a sanity check, run the following commands to verify that the scenario generator, report server, fuzzer, and database are configured correctly. The example uses the SameSite policy on Chrome.

```
# start the report server
cd server/samesite/corpus && docker compose up

# generate depth-1 scenarios
python fuzzer/user_scenario_gen.py ^
    -s samesite -b chrome -d 1

# drive the browser
python fuzzer/fuzzer.py ^
    -s samesite -b chrome
```

Expected outcome. The scenario generator should print one `[+] Saved:` message for each generated scenario. Chrome should launch automatically and execute the generated interactions, eventually terminating with `[FIN] Fuzzing is over`. Successful execution also creates records with `event_type='interaction'` in the `event_entry` table.

Evaluators may stop the execution after confirming that the artifact successfully controls the BUI-level interaction. Completing the entire run is not necessary for this basic functionality test.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Detection of BUI-level policy-enforcement bugs. The artifact reproduces representative policy-enforcement bugs discovered by BUIzz and demonstrates its ability to identify enforcement inconsistencies introduced by BUI-level interactions. Experiments (E1) and (E2) validate this claim.

A.4.2 Experiments

Both experiments are performed using the minimal working example provided in `example/README.md`. The example uses a version-pinned portable Brave browser and a set of pre-configured scenarios that reproduce representative bugs from the paper. Unless stated otherwise, all commands should be executed from the `example/` directory.

One-time preparation. After completing the installation steps in Section A.3.1, run `setup.py` from the `example/` directory. The script downloads the portable Brave browser and prepares a browser profile with the required settings. The same settings can also be applied manually by following the instructions in `example/README.md`.

```
# Download the version-pinned Brave browser
python setup.py
```

Note on the baseline. `scenario_base_line.py` collects the pre-interaction baseline. For Chromium-based targets, the current implementation uses Chrome as the baseline browser, so the script locates the installed Chrome executable even when the target browser is Brave (`-b brave`). A regular Chrome installation must therefore be present in addition to the portable Brave browser prepared by `setup.py`.

(E1): SameSite split-view bug (CVE-2025-48980) [1 human-hour + 1 compute-minute]. This experiment reproduces the SameSite bypass triggered by "Open link in split view' on Brave (Claim C1; Figure 1, Idx #20 in Table 4).

Preparation: Complete the one-time preparation steps in Section A.4.2.

Execution:

```
# deploy the SameSite scenario

python mini_scenario.py 1

# start the SameSite report server

cd ..\server\samesite\corpus
docker compose up -d
cd ..\..\..\example

# collect the baseline
```

```
python scenario_base_line.py ^
-b brave -s samesite

# run OS-level interaction simulation

python ..\fuzzer\fuzzer.py ^
-s samesite -b brave

# compare pre/post-interaction behavior

python ..\analyzer.py ^
-s samesite -b brave

# group inconsistencies into bugs

python ..\deduping.py ^
-s samesite -b brave
```

Expected outcome: The analyzer prints a message such as `[strict] flagged N/M interaction rows` and writes `bugs/samesite/interaction_diff_brave.txt`. The deduplication step should report a distinct bug such as `(Open link in split view, , https:)`. The flagged record contains `violation: strict`, indicating that a `SameSite=Strict` cookie was sent on a cross-site navigation where it should have been blocked.

(E2): CSP split-view bugs [1 human-hour + 1 compute-minute]. This experiment reproduces CSP bypasses triggered by opening `blob:` or `data:` documents in split view on Brave (Claim C1; Idx #7 and Idx #6 in Table 4).

Preparation: Complete the one-time preparation steps in Section A.4.2. In addition, shut down the SameSite report server started in (E1) and remove the Docker containers and images it left behind before starting (E2). Resources from the SameSite environment interfere with the CSP test servers, so (E2) must be started from a clean Docker state.

Execution: For the `blob:` case, run:

```
python mini_scenario.py 2

cd ..\server\csp
docker compose up -d
cd ..\..\example

python scenario_base_line.py ^
-b brave -s csp1

python ..\fuzzer\fuzzer.py ^
-s csp1 -b brave

python ..\analyzer.py ^
-s csp -b brave

python ..\deduping.py ^
-s csp -b brave
```

For the `data:` case, repeat the same commands after replacing `mini_scenario.py 2` with `mini_scenario.py 3`.

Expected outcome: After "Open link in split view", the nonce-less script in the `blob:` or `data:` document executes, indicating that CSP was not inherited or that the navigation was improperly allowed.

Because BUIzz drives a real browser via OS-level input, do not move the mouse or switch windows while the fuzzer is running; the exact flagged counts may vary slightly across runs, but the distinct bugs should still be reproduced.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix – The State of Passkeys: Studying the Adoption and Security of Passkeys on the Web

Louis Jannett [1], Andreas Mayer [2], Maximilian Westers [2], Vladislav Mladenov [1], Christian Mainka [3], and Jörg Schwenk [1]

[1]Ruhr University Bochum
[2]Heilbronn University of Applied Sciences
[3]University of Wuppertal

A Artifact Appendix

A.1 Abstract

We analyze the landscape and security of passkeys on real-world websites using two tools: (1) PASSKEYS-RADAR aggregates multiple sources to identify passkey-enabled websites (§3). We manually registered passkeys on 208 sites and analyzed supported features and passkey management (§4). (2) PASSKEYS-ATTACKER automates security tests after manual account setup and login triggering (§5). We semi-automatically tested 103 passkey-enabled sites (§6).

Our artifacts comprise three components:

- **PASSKEYS-RADAR** aggregates passkey community directories and Internet Archive data, merges them into a unified list of passkey-enabled sites, and presents the results via a web UI (`./radar`). A separate DETECTOR module independently scans 18M CrUX domains (`./detector`) using a parallel-execution framework (`./detector/taskly`, not part of our contribution); we contribute the scanning tasks (`./detector/tasks`). All generated data since 2021 is available in `./data`.
- **PASSKEYS-ATTACKER** is a web-based tool with a frontend (`./tools/frontend`), backend (`./tools/backend`), and Chrome extension (`./tools/extension`). For ethical reasons and because real-world websites are dynamic and many issues were fixed after responsible disclosure, the evaluation does not attempt to reproduce the reported vulnerabilities. Instead, we implemented an intentionally vulnerable learning platform for artifact evaluation (`./learning`) that covers all vulnerabilities from Table 2 and enables safe, controlled experimentation with PASSKEYS-ATTACKER.
- **EVALUATION RESULTS** are processed and visualized in Jupyter notebooks (`./notebooks`), reproducing the figures and numbers from the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

PASSKEYS-ATTACKER simulates a WebAuthn client and authenticator that deliberately skips security checks, enabling fine-grained manipulation of WebAuthn messages throughout the protocol. Therefore, it must be used only in a dedicated testing browser (profile) and with accounts created exclusively for testing. Do not use it in production or with private passkeys (e.g., for your banking account). Use it responsibly and do not exploit vulnerabilities against accounts owned by others. Although most artifacts run in Docker, we recommend an isolated virtual machine for maximum containment.

A.2.2 How to access

- All artifacts are available on GitHub: https://github.com/RUB-NDS/state-of-passkeys-artifacts.
- Releases are automatically archived on Zenodo; the latest release is available at https://doi.org/10.5281/zenodo.17898769.
- This evaluation uses version 4, archived at https://doi.org/10.5281/zenodo.18418005.
- The actively developed PASSKEYS-ATTACKER is also maintained separately for ongoing and future work: https://github.com/RUB-NDS/passkeys.tools. This is in addition to the version included in the main artifacts repository.
- Project website: https://passkeys.tools.
- Public instances are available for PASSKEYS-RADAR (https://radar.passkeys.tools), PASSKEYS-ATTACKER (https://attacker.passkeys.tools and https://db.passkeys.tools), and the learning platform (https://learning.passkeys.tools).
- The latest Chrome extension is also available via https://attacker.passkeys.tools/extension.zip.

A.2.3 Hardware dependencies

Our artifacts run on any machine capable of running a modern GUI-based OS (e.g., `Ubuntu 24.04 LTS`) with Docker, Chrome, Python 3, and Pipenv installed.

A.2.4 Software dependencies

Our artifacts should run on any modern GUI-based OS (x86 or ARM) that is capable of running Docker, Chrome, Python 3, and Pipenv. We tested and recommend `Ubuntu 24.04 LTS` or `macOS Tahoe 26.2` (on the host or in a VM). Windows 11 is untested but should work as well.

A.2.5 Benchmarks

None.

A.3 Set-up

While newer versions should also work, we refer to specific versions that we have used to verify our artifacts.

A.3.1 Installation

Installation:

- **Install Docker**: Follow the official installation guide at https://docs.docker.com/get-started/get-docker/. Then, run `docker --version` and ensure that version `29.1.5` (or newer) is installed and working. Also run `docker compose version` and verify that version `v5.0.1` (or newer) is installed and functional.
- **Install Chrome**: Download and install Chrome from https://www.google.com/chrome. After installation, open Chrome, navigate to `chrome://version`, and confirm that version `144.0.7559.97` (or newer) is installed.
- **Install Python**: Install Python `3.14.2` via https://www.python.org/downloads/release/python-3142/ (or using a package manager of your choice). Verify the installation by running `python3 --version` and confirming it reports `3.14.2`.
- **Install Pipenv**: Install Pipenv `2026.0.3` by following https://pipenv.pypa.io/en/latest/installation.html. We recommend using your system package manager (e.g., `apt` or `brew`) where available. Verify the installation by running `pipenv --version` and confirming it reports `2026.0.3`.
- **Clone the repository**: Clone the repository using `git clone https://github.com/RUB-NDS/state-of-passkeys-artifacts.git`, or download and unpack the artifact bundle from Zenodo: https://doi.org/10.5281/zenodo.18418005.
- **Install Pipenv dependencies**: Change into the notebook directory with `cd ./notebooks` and run `pipenv install` to install all dependencies.

Configuration:

- **Configure Chrome profiles**: Open Chrome and click the profile icon. Select "Manage Chrome profiles" and create two new, clean browser profiles named "Victim" and "Attacker". This setup emulates two isolated browser instances and keeps the experiment separate from potential existing personal or work profiles.
- **Configure the Chrome extension**: In *both* profiles, open `chrome://extensions`, enable "Developer mode" (top right), click "Load unpacked", and select the `./tools/extension` directory. Then, open the extension popup via the extension icon next to the address bar. Ensure the extension is enabled and that the Frontend URL is set to the default https://passkeys.tools. For the operation mode, select `Profile 1` in the victim profile and `Profile 2` in the attacker profile.
- **Configure PASSKEYS-ATTACKER**: In the victim profile, open https://passkeys.tools/settings and select "Remote Storage". Verify that the "Remote Server URL" is set to the default https://db.passkeys.tools. Choose a secret (or generate one) and enable end-to-end encryption. Click "Test Connection" and confirm that the message "Connection successful!" is displayed. Finally, click "Save Storage". Repeat the same steps in the attacker profile, but make sure to use the *same* secret as in the victim profile.
- **Configure the learning platform**: In the victim profile, open https://learning.passkeys.tools, click "Instance", and copy the UUID shown under "Current Instance". Then, in the attacker profile, open https://learning.passkeys.tools, click "Instance", paste the victim UUID into "Switch Instance", and click "Switch".

A.3.2 Basic Test

Validate PASSKEYS-ATTACKER: In the victim profile, open https://learning.passkeys.tools and create a new account. Then, go to https://learning.passkeys.tools/settings, click "Add Passkey", and (optionally) enter a name. Select "Register Passkey"; the PASSKEYS-ATTACKER interceptor should open, display the WebAuthn request, and generate a valid response. Click "Send" and confirm that the new passkey appears in the account settings.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PASSKEYS-RADAR and the DETECTOR module (§3.1) aggregated 8,523 websites from multiple sources (§4.1). After merging (§3.2), 872 unique sites remained (§4.1). We excluded 664 sites from further analysis (Table 3) because manual account registration was feasible on only 208 sites (§4.2). We then manually analyzed these 208

sites, including their modes and algorithms (Figure 6). While we tracked passkey-enabled sites over time (Figure 5), the manual analysis is based on the snapshot retrieved on April 22nd, 2025 (§4.2). These findings are supported by experiments (E1), (E2), and (E4).

(C2): Due to practical restrictions, we could only fully analyze passkey implementations on 103 websites (§6.1) using PASSKEYS-ATTACKER (§5.2, §5.3), which automates 28 detection methods for 15 attack types (Table 2). For ethical reasons, and because real-world websites are dynamic and many issues were fixed after responsible disclosure, we demonstrate PASSKEYS-ATTACKER only in a controlled lab setting. To this end, we implemented an intentionally vulnerable learning platform for artifact evaluation that covers all vulnerabilities (Table 2). We provide step-by-step guidance to identify and exploit one selected vulnerability, namely *Swap cred*^{id}. These findings are supported by experiments (E3) and (E5).

A.4.2 Experiments

Experiments (E1)–(E3) prove the functionality of our artifacts. (E4) and (E5) focus on the reproducibility of our artifacts.

(E1): Identify passkey-enabled websites with PASSKEYS-RADAR *(1 human-hour + 0.1 compute-hour)*

Preparation: Deploy an empty PASSKEYS-RADAR instance by following `./radar/README.md` ("Production Deployment" and "Docker Run"). Open `http://localhost:8090` and confirm the UI loads (the initial "no data" error is expected).

Execution: In "Admin" → "Operations" → "Select & Fetch", click "Select All Directories" and then "Fetch Selected". Authenticate with "admin" and the password from `./radar/.env`. After all fetches complete, verify in "Task History" that all 12 directories were fetched. Then run the pipeline: "Operations" → "Combine" → "Merge" → "Select Unmerged" → "Merge Selected".

Results: In "Admin" → "Data Browser" → "Merged", open the newest file via "View". It should contain a deduplicated list of passkey-enabled sites plus the directories in which each site appears. In "Explore", select the next day (after the merge run) and click "Load Data" to view the merged list as an interactive table. Note that (E1) covers directory sources only; well-known discovery is handled in (E2). Compare "Overview" → "Domains by Directory" on your local instance (`http://localhost:8090`) with our public instance (`https://radar.passkeys.tools`, last updated August 27th 2025). You should observe slightly higher counts in your run, reflecting websites that have recently adopted passkeys. After artifact evaluation, we will refresh the public instance with an updated dataset.

(E2): Identify well-known documents with the DETECTOR module *(0.5 human-hour + 0–1 compute-hour)*

Preparation: Run `cd ./detector` and follow `./detector/README.md` ("Build" and "Run"). Verify that `cli.py` runs successfully and prints the help output.

Execution: Run the "Scan a single origin" command from `./detector/README.md` to scan `kayak.com` for well-known files. The scan should finish within a few seconds to about a minute. Optionally, use the provided command to scan the top 1k CrUX origins; this may take $\geq$1 hour depending on your Internet connection.

Results: After completion, `./detector/output/wellknown/<scan_id>/<task_id>/result.json` should be created and contain results for the scanned origin. For `kayak.com`, both documents (`/.well-known/passkey_endpoints` and `/.well-known/webauthn`) should be present and non-empty. We also include our recurring scans of well-known documents for 18M CrUX domains in `./data/wellknown`.

(E3): Identify the *Swap cred*^{id} vulnerability with PASSKEYS-ATTACKER *(1 human-hour + 0 compute-hour)*

Preparation: Set up the victim and attacker profiles as in §A.3.1. On `https://learning.passkeys.tools`, open "Instance" and click "Reset" to start fresh.

Execution: *In victim profile:* Create a victim account → account settings → select "Duplicate Cred ID" security verifier → "Add Passkey" → "Register Passkey". Keep the default "Credential ID" and "Key" in the interceptor ("Actions" → `profile1 | ES256`), then click "Send" to complete registration.

In attacker profile: Create an attacker account → account settings → select "Duplicate Cred ID" security verifier → "Add Passkey" → "Register Passkey". Select `profile1 | ES256` as "Credential ID" in the interceptor, but keep the default `profile2 | ES256` "Key", then click "Send" to complete registration. This binds the attacker key to the victim credential ID, creating a duplicate ID.

Results: *Victim lockout:* Switch to the victim profile and log out. On the login page, select the "Discoverable" demo verifier and click "Login with Passkey". Use `victim` as "User Handle" and `profile1 | ES256` for "Credential ID" and "Key", then click "Send". Verify the error message and that the victim cannot log in.

Recovery: In the attacker profile, delete the attacker passkey from account settings. Repeat the victim login attempt and confirm the victim can log in again.

(E4): Landscape evaluation results *(0.5 human-hour + 0.1 compute-hour)*

Preparation: Run `cd ./notebooks` and execute `pipenv run jupyter notebook`. This should open the Jupyter UI in your browser.

Execution and Results: *Adoption of passkeys (Figure 5)*: Open `radar.ipynb` in the Jupyter UI and click "Restart the kernel and run all cells". The notebook reads `./data/merged` to generate Figure 5 and additional charts not included in the paper.

Table 1: Mapping of `sheet.ipynb` cells to paper sections

Cell	Paper Reference
5	§6.2.7
6	Table 3, §4.2 (“Intro”)
7	§4.2 (“Confirming Passkey Registrations”)
8	§4.2 (“Removing Passkeys”)
10	§4.2 (“Removing Passkeys”)
11	Figure 6a
12	§4.2 (“Passkey Authentication Modes”)
13	Figure 6b
14	§4.2 (“Passkey Algorithms”)
15	§4.2 (“Passkey Scopes”)
18	§4.2 (“User Identifier”)
21	§4.2 (“Key Attestation”)
22	§4.2 (“Authenticator Selection”, “User Verify”)
24	§6.2.*, Table 2
27	§6.2 (“Attack Severity”)
28	Figure 8b
29	Figure 8a

Fetching all sources: Our landscape and security analyses use the snapshot generated on April 22nd, 2025. Verify the aggregated 8,523 websites with: `jq '[.directories[], .wellknown[]] | map(length) | add' 2025-04-22-12-36-53-combined.json`.

Merging all sources: Verify that merging yields 872 unique sites with: `jq 'length' 2025-04-22-12-36-53-merged.json`.

Manual evaluation sheet: We manually visited all 872 sites to attempt account creation and passkey registration. The results are provided as `sheet.xlsx` and `sheet.csv`, containing all domains from `2025-04-22-12-36-53-merged.json` sorted by Tranco rank. Column “Availability” records whether a site was analyzable and, if not, the failure reason. For example, `sheet.xlsx` contains 10 entries labeled “Payment method required”, matching Table 3.

(E5): Security evaluation results *(0.5 human-hour + 0.1 compute-hour)*

Preparation: Same as (E4).

Execution and Results: In the Jupyter UI, open `sheet.ipynb` and click “Restart the kernel and run all cells”. The notebook reads `sheet.csv` and reproduces most numbers and figures used in the paper. Table 1 maps notebook cells to the corresponding parts of the paper.

Example 1: Cell 24 should output “Test Create Context Nonsense: 5”, corresponding to Table 2 (*Context* → *Nonsense* → Reg. (Create)), i.e., 5 vulnerable sites.

Example 2: Cell 15 should output “Upscoped: 80”, corresponding to “80 sites (38%) explicitly scope their passkeys to the top-level domain” (§4.2).

A.5 Notes on Reusability

Both PASSKEYS-RADAR and PASSKEYS-ATTACKER are designed as extensible platforms for ongoing passkey research. PASSKEYS-RADAR can serve as a longitudinal observatory for tracking passkey adoption trends. Researchers can add new community directories, integrate additional data sources, or adapt the DETECTOR module to scan for emerging well-known document formats as the WebAuthn ecosystem evolves. PASSKEYS-ATTACKER provides a modular framework for security analysis: its interceptor architecture allows researchers to implement new attack vectors. A promising direction for future work is fully automating account creation and passkey registration workflows, which would enable large-scale security assessments by integrating PASSKEYS-ATTACKER. Similarly, developers can embed PASSKEYS-ATTACKER into their own unit and integration tests to continuously verify that their WebAuthn implementations correctly enforce security checks. Contributions to both tools are welcome via GitHub.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Meerkat: Pushing the Practical Limits of Dynamic Bisection with PoC Mutation

Joseph Bursey
University of California, Irvine

Christoph Sendner
University of California, Irvine

Ardalan Amiri Sani
University of California, Irvine

Zhiyun Qian
University of California, Riverside

A Artifact Appendix

A.1 Abstract

This appendix is provided to confer clear instructions in order to run and evaluate Meerkat, our dynamic bisection tool which utilizes PoC mutation during bisection. Using our instructions and scripts, our readers should be able to verify the claims made in the paper that Meerkat is able to reach the BiC in 34 more cases than Syz-bisect, and that it is able to do so in a comparable runtime.

A.2 Description & Requirements

The presented repository contains the entire source code for Meerkat, the dataset and results from our paper (`results/`), a fork of Syzkaller (`syzkaller/`), and scripts to build a Debian Stretch image (`image/stretch/`), as well as easy-to-use scripts to build and run everything from a single command. The setup script additionally pulls a set of compilers to be used during bisection from a separate Zenodo repository.

Meerkat has been tested on Ubuntu 20.04 and Ubuntu 24.04. It will likely run on other debian-based systems, but has not been tested. Meerkat should be run on and x86_64 machine, and can be run in a VM assuming nested virtualization is enabled. Meerkat uses 16 threads (`nproc` $>=$ 16) to build the kernel and boot VMs, but 20 is recommended. Lastly, Meerkat allocates up to 32GB of memory across 8 VMs, so the host system should have at least that much to prevent bottle necks.

A.2.1 Security, privacy, and ethical concerns

Meerkat does not pose any privacy or security-related risks to its users. Meerkat does not export any information from the host system, nor does it remove any safeguards from the system. All code and libraries are pulled from trusted sources. All other ethical concerns are discussed in the Ethical Considerations appendix.

An informational note: the `artifact-verify.sh` script adds the current user to the `kvm` group for the purpose of booting qemu VMs. Additionally, the `artifact-verify.sh` script creates a symbolic link for the missing library `libmpfr.so.4`, simply pointing it at the newer `libmpfr.so.6`. This is for the purpose of making the older compilers function on newer systems.

A.2.2 How to access

Meerkat has been made accessible on both Zenodo: https://doi.org/10.5281/zenodo.20317981, as well as GitHub: https://github.com/trusslab/meerkat. The compiler repository can be accessed on Zenodo as well: https://doi.org/10.5281/zenodo.20316000.

A.2.3 Hardware dependencies

Meerkat builds the Linux kernel and boots several VMs simultaneously. For this it expects a CPU capable of at least 16 threads (`nproc` $>=$ 16), as well as at least 32GB of memory. The Kernel repository and images can be rather large as well. We recommend around 32GB of free disk space.

A.2.4 Software dependencies

Meerkat is provided with easy-to-use scripts which will download and install all known dependencies. These include:

- Several packages pulled from `apt`: make, git, gcc, g++, ccache, build-essential, flex, bison, libncurses-dev, libelf-dev, libssl-dev, dwarves, libdw-dev, qemu-system-x86, libmpfr-dev, debootstrap, dpkg.
- A set of compilers pulled from our Zenodo repository.
- A recent version of Go.

A.2.5 Benchmarks

The 200 bug dataset is provided in the repository (`parse/bugs.csv`).

A.3 Set-up

Our artifact evaluation script (`artifact-verify.sh`) automatically sets up the environment, handling all dependencies to the best of our knowledge. Should the user desire to setup up the environment before running Meerkat, they may use the `setup.sh` script.

A.3.1 Installation

The `setup.sh` script discussed above handles all installation of Meerkat's dependencies. Meerkat itself does not need to be installed, merely built.

A.3.2 Basic Test

A basic functionality test can be conducted using:

```
./artifact-verify.sh functional
```

This will bisect a single bug that should complete in around 14 hours. Alternatively, if the evaluator does not want to bisect an entire bug at this stage, they may use the command:

```
./mk-manager.sh -s 2 -e 2 -i 1 \
-b parse/bugs.csv \
-F ff-test,poc-all-pocs,find-only
```

This will build and test at the anchor commit, and should complete in around 45 minutes. Documentation on `mk-manager.sh` can be found at `docs/mk-manager.md`.

As the output from Meerkat is quite long, please refer to our documentation provided in the repository at `docs/reading-results.md`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Meerkat is able to reach the BiC in 34 more cases than Syz-Bisect.

(C2): Meerkat completes bisection in a total runtime that is closely comparable to Syz-bisect.

A.4.2 Experiments

The experiments carried out in our paper took several months to complete computationally, as well as a considerable amount of human-hours to examine. The entire experiment can be recreated using the provided scripts, however we do not expect our users to do so. Instead, we provide a subset consisting of the 37 bugs where Meerkat was able to reach the BiC, but Syz-bisect could not (The difference between 34 and 37 is explained in `docs/README.md`). This should be enough to verify the result in **(C1)**, while not taking several months to complete.

(E1): [17+ compute-days + 1-2 human-hours]: Run Meerkat on the smaller 37 bug dataset, and observe several results from the bisection logs.
How to: Execute our `./artifact-verify.sh` script with the described preset, allow it to run for about 2 weeks, then read the results from the bisection logs.
Preparation: Optionally run `./setup.sh`. This step is automatically handled by `artifact-verify.sh`.
Execution: Execute the command:
`./artifact-verify.sh reproducible`
Please do this in `screen`/`tmux`. Based on our results, this process should take a little over 2 weeks, but it may take longer depending on the user's environment.
Results (C1): Observe the bisection logs in the files `wd-meerkat-1/log/bug####.log`. Compare the final results marked "Bisection Result:" to the ground truth hashes provided in `results/bics.csv`. In ideally all cases Meerkat has reached the correct BiC, but some variance is expected. You can further compare to the Syz-bisect results provided in `results/sb/`. The bisected hash is provided on the final line of each file.
Results (C2): Observe the same log files from above, this time noting the "Run Time:" result. It is recommended to export the times to a spreadsheet, formatting as a duration. Observe that the average and median runtimes are similar to what is presented in the paper. You can further compare against the provided Syz-bisect results. Grep for "total time:" to find the runtime in the log.

(E2): [several compute-months + several human-days]: Run the entire ablation study. We do not expect you to run this. This is only here to provide instruction if a reader requires it.
How to: Execute our `./artifact-verify.sh` with multiple presets in parallel, allow to run, then read the results from the log files.
Preparation: Run `./setup.sh`. This will stop parallel instances of `./artifact-verify.sh` from colliding at this step.
Execution: Execute the commands:
```
./artifact-verify.sh mk full
./artifact-verify.sh mkm1p full
./artifact-verify.sh mknm full
./artifact-verify.sh mk1p full
```
Run each instance in parallel when possible and spread across multiple computers/servers when possible.
Results: Observe the results in the same way as **(E1)**, accounting for different working directories.

A.5 Notes on Reusability

Extensive documentation is provided in our code repository, both on Zenodo and GitHub. Please refer to `docs/`.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: BenchChecker

Di Wu, Xu He, Shu Wang, and Kun Sun

A Artifact Appendix

A.1 Abstract

BenchChecker is an artifact for assessing data contamination in LLM bug-fixing benchmarks. It implements the paper's two-stage workflow: Stage I probes a target model for repository-presence signals using signature query/target pairs, and Stage II checks patch presence by comparing benchmark patches with model-cutoff repository snapshots, using GumTree-backed localization, embedding similarity, replacement-diff construction, and optional container test-suite verification. The artifact also includes packaged calibration data, benchmark loaders, tests, and evaluator-facing documentation. The final artifact received the Artifacts Available and Artifacts Functional badges.

A.2 Description & Requirements

The final evaluated artifact is BenchChecker-AE v1.1.7. It contains the Python package, command-line interface, benchmark loaders, Stage I and Stage II implementations, unit tests, documentation, and vendored external tools: NiCad 6.2, FreeTXL for Linux-native NiCad runs, and GumTree 4.0.0-beta4. The release also includes a Dockerfile for running BenchChecker without installing its Python and Java dependencies directly on the host. Version 1.1.7 filters invalid empty Stage I query/target pairs before model probing and defensively rejects an empty prompt. This addresses the zero-length Qwen input reported during evaluation. The main AE entry point is:

```
python -m benchchecker.cli full \
  BENCHMARK CASE_ID TARGET_MODEL
```

The command clones the benchmark repository into a workspace, resolves the target model cutoff snapshot, runs both stages, and writes JSON/JSONL outputs under `workspace/runs/`.

A.2.1 Security, privacy, and ethical concerns

The artifact is not destructive and does not intentionally disable security mechanisms. It may clone public repositories, download public HuggingFace models/datasets, install Python packages, and run benchmark test suites in Docker containers when configured. Evaluators should run it in a normal isolated artifact-evaluation workspace and should not place secrets in that workspace except an optional `HF_TOKEN` if their HuggingFace access requires one. Full verification from the supplied container can mount the host Docker socket; this gives the container control over the host Docker daemon. Use the documented restricted mode if that access is not acceptable. No author-owned hardware, reviewer identity, or personally identifying information is required.

A.2.2 How to access

The final stable URL is the permanent Zenodo concept DOI:

`https://doi.org/10.5281/zenodo.20385622`

This DOI represents all versions and resolves to the newest published one. The final file is `BenchChecker-AE-v1.1.7.zip`. Its SHA-256 is:
`73a79296bb935cf349fc98600c4246a58`
`b5145f5774fcf10a66069c25f6aa1d4`

A.2.3 Hardware dependencies

No specialized hardware is required for the functional smoke tests; the `tiny` model profile can run on CPU. For full-scale runs with real target models and faster embedding similarity, we recommend Linux x86_64 with an NVIDIA CUDA GPU and 20–50 GB free disk for repository clones, model caches, and Docker runs. Reviewers can use evaluator-owned compute; no remote author-owned platform is needed.

A.2.4 Software dependencies

Recommended: Ubuntu 22.04 or 24.04 LTS, Python 3.10 or 3.11, Git, Java 17 or newer for GumTree, Docker Engine with a running daemon for container verification, and a C/C++ build toolchain only as required by Python wheels or benchmark dependencies. Windows 11 and macOS are supported for source-level runs, but Docker Desktop is required for containerized benchmark tests and NiCad may use the Docker fallback on non-Linux hosts. Python dependencies are declared in `pyproject.toml`; the optional `artifact` extra installs the SWE-bench harness. The supplied Dockerfile provides a fixed Python 3.11, Java 17, NiCad, FreeTXL, GumTree, and Docker CLI environment.

A.2.5 Benchmarks

The artifact supports network-backed loaders for SWE-bench, SWE-bench Verified, SWE-bench Pro, and Multi-SWE-bench, and path-backed loaders for BugsInPy, BugSwarm, Defects4J,

GrowingBugs, and local JSON/JSONL exports. The archive also includes the original v1.0.0 label and word-frequency assets in `data/`, documented in `data/README.md`.

A.3 Set-up

A.3.1 Installation

Download and unpack the Zenodo archive, then run:

```
python -m venv .venv
. .venv/bin/activate
python -m pip install -e ".[dev,artifact]"
python scripts/install_nicad.py
python -m benchchecker.cli full --help
```

On Windows, use `.venv\Scripts\activate` instead of the POSIX activation command. To inspect Docker/container readiness:

```
python -m benchchecker.cli env doctor
python -m benchchecker.cli env bootstrap \
  --dry-run
```

Alternatively, build and check the supplied CPU container:

```
docker build -t benchchecker-ae:1.1.7 .
docker run --rm benchchecker-ae:1.1.7 --help
```

The restricted and host-Docker modes, GPU option, persistent caches, and mount-path mapping are documented in `docs/CONTAINER.md`.

A.3.2 Basic Test

Run the full workflow with a tiny local model:

```
python -m benchchecker.cli full \
  swebench psf__requests-1724 tiny \
  --workspace workspace/ae-smoke \
  --verification-backend swebench \
  --stage1-max-queries 3
```

Expected outputs include a run directory with a JSON report and generated Stage I pair/probe files. If Docker or the SWE-bench harness is unavailable, the run still records the explicit container status in the report rather than silently hiding the environment issue.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): BenchChecker provides an executable implementation of the two-stage contamination-detection pipeline described in the paper: repository presence followed by patch presence (paper Sections 3–4; E1–E3).

(C2): Across six bug-fixing benchmarks, BenchChecker achieves AUC around 0.9 while keeping the false-positive rate at or below 0.1 (paper Section 5; E4).

(C3): Filtering detected contamination reduces the top-10 systems' reported resolution rates by more than 20% on medium-difficulty tasks in SWE-bench Verified and Multi-SWE-bench (paper Section 6; E4).

A.4.2 Experiments

(E1): Environment and package check, 5–10 human-minutes.

Run `python -m benchchecker.cli full -help`, `python -m benchchecker.cli env doctor`, and optionally `pytest -q`. These commands check that the CLI, package data, vendored tools, and local environment are visible.

(E2): Functional smoke run, 10–30 human-minutes plus compute time.

Run the Basic Test command above. The expected result is a completed run directory with `report.json`, generated Stage I query/probe files, and explicit Stage II verification status. This exercises the main end-to-end code path without requiring a large target model.

(E3): Real-model single case, evaluator-dependent runtime.

For a stronger functionality check, run:

```
python -m benchchecker.cli full \
  swebench django__django-13023 \
  qwen2.5-coder-0.5b \
  --workspace workspace/ae-qwen \
  --verification-backend auto \
  --stage1-max-queries 50
```

The expected result is a populated report for a real model profile. For models not built into `benchchecker/models.py`, provide `-cutoff-date YYYY-MM-DD`. In v1.1.7, generated Stage I pairs have non-empty prompts and targets, so Qwen is not called with a zero-length input.

(E4): Optional reproduction analyses.

Use the reproduction guide, evaluation notebooks, and utility scripts to rerun Stage I, Stage II, baseline, and leaderboard analyses. These runs require the corresponding benchmark data, model access, and substantially more compute than the smoke tests.

A.5 Notes on Reusability

BenchChecker can be used beyond the paper by selecting a supported benchmark loader or by providing a local JSON/JSONL benchmark export with repository, patch, commit, and cutoff metadata. The language-specific preprocessing and configuration files are split under `benchchecker/stage2/preprocess/` and `benchchecker/configs/`, so reviewers can inspect or extend the artifact to additional code-generation benchmarks.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: VeCT: Secure and Efficient Constant-Time Code Rewriting with Vector Extensions

Qisheng Jiang
Duke University
qisheng.jiang@duke.edu

Danfeng Zhang
Duke University
danfeng.zhang@duke.edu

A Artifact Appendix

A.1 Abstract

The artifact provides the source code and experimental framework for VeCT, a compiler-based code rewriter that leverages vector extensions to retain constant-time guarantees while improving performance. It includes the VeCT core tool, a t-test framework for side-channel leakage detection, microbenchmarks and real-world applications for performance evaluation and security validation, and all scripts and data required to replicate the t-tests and performance measurements.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All experimental evaluations should be conducted within a controlled and isolated environment, ensuring no risk to external systems or user data. VeCT is designed as a defensive tool to assist researchers and developers in identifying and mitigating timing side-channel vulnerabilities. Additionally, we provide a Dockerfile to facilitate a containerized environment, which ensures isolation during the artifact evaluation process.

A.2.2 How to access

Our tool and all benchmarks used in our experimental evaluation are publicly available at `https://doi.org/10.5281/zenodo.17822446` and `https://github.com/qisheng-jiang/VeCT`. You may refer to `Evaluation.md` for the artifact evaluation. Please refer to the GitHub repository for the latest update.

A.2.3 Hardware dependencies

The artifact requires an Intel CPU with support for the **AVX-512** instruction set. Users can verify hardware compatibility by checking the CPU flags: `lscpu | grep avx512`. We performed experiments on a machine with an Intel Core i9-11900 and 32 GB DRAM running Ubuntu 24.04.

A.2.4 Software dependencies

We recommend using Docker to ensure a consistent environment. The provided environment is based on the `gcc:7.5.0`. Key software requirements include:

- LLVM and Clang.
- Build tools: `cmake`, `make`, and `ninja-build`.
- Python 3 with `pandas`, `matplotlib`, `numpy`, `click`, and `seaborn` libraries.

Please refer to the `Dockerfile` for more details.

A.2.5 Benchmarks

The evaluation utilizes several benchmarks:

- **T-test Analysis:** Statistical analysis for AVX-512 memory access instructions.
- **Microbenchmarks:** Targeted tests for performance and security validation.
- **Real-World Applications:** Benchmarks from open-source real-world applications.

A.3 Set-up

A.3.1 Installation

(I1): Build the Docker image:

```
docker build -t image-vect .
docker run --rm -it -v "$(pwd)":/app image-vect /bin/bash
```

(I2): Inside the container, compile and install VeCT:

```
cd /app/src
./install.sh
. ./setup.sh
./llvm_compile_dfsan_cpp.sh
apt update && apt upgrade -y
apt install -y llvm-13 clang-13
cd /app/src/passes && make install -j10
cd /app/src/lib && make install -j10
```

(I3): Compile and install baselines for comparison:

```
cd /app/src_constantine+
. ./setup.sh
cd /app/src_constantine+/passes && make install -j10
cd /app/src_constantine+/lib && make install -j10
```

A.3.2 Basic Test

To verify the installation, run the following script:

```
cd /app/src
bash run.sh microbenchmark/vector-perf.c
./microbenchmark/vector-perf.out < ./real-world-apps/binsec/random_input.txt
```

Successful execution should produce the similar output:

```
Success: microbenchmark/vector-perf.out
Elapsed time in microbenchmark/vector-perf.c: [2026]
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Constant-time guarantees for AVX-512 memory access instructions (Section 4).
(C2): VeCT maintains high performance across microbenchmarks and constant-time guarantees for the generated code (Section 6.1).
(C3): VeCT effectively repairs real-world applications without significant overhead (Section 6.2).

A.4.2 Experiments

(E1): *T-test Analysis [40 compute-minutes]:*
Execution:

```
cd /app/t-test
bash t-test.sh
```

Results: `t-test/results.md` shows the reproduced results for Figures 2-5 in Section 4.

(E2): *Microbenchmarks [4.6 compute-hours]:*

(E2.1): *Overview and Impact of Access Count [40 compute-minutes]:*
Execution:

```
# run VeCT (Vector and Single)
cd /app/src/microbenchmark
bash run_microbenchmarks.sh
# run baselines (Constantine+ and Original insecure code)
cd /app/src_constantine+/microbenchmark
bash run_microbenchmarks.sh
# generate results
cd /app/src/microbenchmark
python3 stats_pic.py
```

Results: `src/microbenchmark/results.md` shows the reproduced results for Figures 15 and 16 in Section 6.1.

(E2.2): *Security Validation [4 compute-hours]:*
Execution:

```
cd /app/src/microbenchmark
bash run_validation.sh
```

Results: `src/microbenchmark/validation.md` shows the reproduced results for security validation in Section 6.1. The output should look like the following for each setup:

```
Processing groups: [xxx] with masks: index 0, index 1
meas: [0.00] M, max t:   [+0.30], max tau: [6.71e-02], (5/tau)^2: [5.56e+03]. For the moment, maybe constant time.
```

(E3): *Real-World Applications [35.4 compute-hours]:*
Execution:

```
# run VeCT (Vector and Single)
cd /app/src/real-world-apps
bash run_app_tests.sh
# run baselines (Constantine+ and Original insecure code)
cd /app/src_constantine+/real-world-apps
bash run_app_tests.sh
# generate results
cd /app/src/real-world-apps
python3 stats_table.py
```

Results: `src/real-world-apps/results.md` shows the reproduced results for Table 1 in Section 6.2.

A.5 Notes on Reusability

VeCT is designed with modularity to facilitate adoption across different hardware architectures and software environments:

- **Hardware Portability:** While evaluated on Intel AVX-512, the t-test framework can be ported to other architectures with low effort by replacing architecture-dependent instructions for vector operations, cache flushes, and time measurements.
- **Toolchain Integration:** Our proof-of-concept is decoupled into four components. While our prototype integrates with Constantine for violation detection and control-flow linearization, these dependencies are not strictly required. Researchers can replace these modules with other tools with moderate engineering effort.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: CULPA: Universal Detection of Memory-Safety Bugs in Unsafe Rust Through the Lens of Safety Requirements

Hung-Mao Chen, Bo Lu, Xu He, Xiaokuan Zhang, Kun Sun

A Artifact Appendix

A.1 Abstract

In this paper, we present CULPA, a universal detector for memory-safety bugs in Rust programs. The key insight of CULPA is to detect the root cause of such bugs: the violation of safety requirements in unsafe Rust contexts. CULPA covers existing bug classes addressed by four static analyzers and identifies additional memory-safety vulnerabilities beyond their scope. We evaluate CULPA on the top 1,000 Rust packages. CULPA uncovers 55 previously unknown (zero-day) memory-safety bugs, 29 of which have been confirmed by developers. Most of these vulnerabilities are missed by four state-of-the-art Rust static analyzers and one LLM-based tool. The data artifacts of this paper include source code, detected bugs, and assigned RustSec/CVE IDs.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We take ethics seriously in this project. All Rust repositories we tested in the paper are publicly accessible on Github. During evaluations of our work, CULPA identified several previously unknown vulnerabilities in widely used software. In each case, we followed a responsible disclosure policy, and reported our discovered vulnerabilities to the developers. We also submitted our findings to the CVE program and the RustSec Advisory Database. We did not disclose those issues to anyone else. All the examples mentioned in the paper are the issues that have been acknowledged and fixed. The RustSec IDs are: RUSTSEC-2025-0137, RUSTSEC-2025-0140, RUSTSEC-2025-0143, RUSTSEC-2026-0001, RUSTSEC-2026-0008; The CVE ID is CVE-2026-0810.

A.2.2 How to access

The artifact can be downloaded from Zenodo: https://zenodo.org/records/20740755

A.2.3 Hardware dependencies

We deploy CULPA on a server with 48-core Intel Xeon CPU ES-2630 and 256 GB memory. A regular PC can support our tool if the user doesn't run the detector with multiple threads.

A.2.4 Software dependencies

All the software dependencies can be installed automatically in the docker container. If setting up in local environment, users are required to install `rustc` with specific version `1.87.0-nightly` and the components including `rust-src`, `rustc-dev`, `llvm-tools-preview`, `miri`. The instructions of installing software dependencies are provided in the file `README.md`.

A.2.5 Benchmarks

Following datasets are included in our artifact:

First, 1000 Rust packages (`top_1000_crates.txt`), which were downloaded before December 1, 2025.

Second, reports that include all CULPA detection results on 1000 packages (`culpa_findings.csv`): The dataset includes all true positives, false positives, FFI (which is out of our scope and should not be count into true/false positives). Based on same results, we also compare whether four state-of-the-art static analyzers and one LLM-based tool can identify the same bugs (`zero_day_comparison_add_s4u`): "yes" means they can detect the same bugs at same location, "no" means they detect the same bugs.

A.3 Set-up

A.3.1 Installation

We provide docker to automatically install and compile all the dependencies and CULPA into the container.

A.3.2 Basic Test

The basic test can be started by setting up the docker we provide in artifacts. The instructions for running a simple functionality test are provided in the section: *Sample usage* of the file `README.md`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): CULPA can uncover 55 new bugs. This is proven by the experiment E1.

(C2): CULPA can detect more new bugs than baseline tools (4 static analyzers and 1 LLM-based tool). This is proven by the experiment E2.

A.4.2 Experiments

(E1): 10 human-minutes + 10 compute-minutes + 50GB disk

How to: First, verify **Preparation** has been done. Second, follow **Execution** to run the docker script. Third, Compare the **Results** to verify whether it matches the data of `culpa_findings.csv`.

Preparation: Build the docker image with `docker build -t culpa:eval .`.

Execution: Run the docker script with `docker run -it culpa:eval`.

Results: The standard output will show all true positives that should be found and the corresponding number in each crate. It should show "TP (new bug) rows reproduced: 55 / 55" at the end.

(E2): 60 human-minutes + 2 compute-hour + 50GB disk

How to: The source code of the packages on existing RUSTSEC bugs have been included in the directory of `rustsec`. First, follow **Preparation** to install the tools as benchmarks,

Preparation: Create the shell with CULPA by `docker run -it culpa:eval bash`. In the shell, install the five baseline tools: TYPEPULSE, SafeDrop, MirChecker, RUDRA, Safe4U.

Execution: Following the instructions of baseline tools to check whether they can detect the same bugs.

Results: The detection results should match the ones in `zero_day_comparison_add_s4u.csv`.

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://zenodo.org/records/20740755.



USENIX Security '26 Artifact Appendix: Wormholes in the File System: Understanding the Misunderstanding of Symlinks

Yongheng Liu
Fudan University

Lei Zhang *
Fudan University

Yuhang Zhao
Fudan University

Yuzhou He
Fudan University

A Artifact Appendix

This artifact appendix is intended to serve as a self-contained document for evaluating our artifact. It provides a structured roadmap for reproducing the findings in our paper "Wormholes in the File System: Understanding the Misunderstanding of Symlinks".

A.1 Abstract

The artifact includes the source code, datasets, scripts, results and instructions needed to reproduce the main experimental results reported in the paper. It has two main parts. `PathResolution` part reproduces our measurements of file type identification and path normalization interfaces across ten programming languages. `PathTraversalFinding` part contains our archive-testing framework, which constructs ZIP, TAR, and 7z proof-of-concept archives for the bypass and arbitrary-write strategies described in the paper, and evaluates them against archive extraction libraries and tools.

A.2 Description & Requirements

This section lists all the necessity to recreate the same experimental setup we have used to run our artifact.

A.2.1 Security, privacy, and ethical concerns

Conducting the artifact does not raise any security, privacy, or ethical concerns. All experiments are conducted within isolated Docker containers. The vulnerability detection tool generates crafted archive files for testing purposes only and does not perform any destructive operations on the host.

A.2.2 How to access

The artifact is available at our Zenodo repository: https://doi.org/10.5281/zenodo.20325581.

A.2.3 Hardware dependencies

No special hardware is required. Our testing environment is built on an x86_64 Linux server.

*Corresponding author

A.2.4 Software dependencies

The artifact requires Docker to be installed on the Linux system. All other dependencies are encapsulated within the provided Docker images. No additional software installation is needed on the host.

A.2.5 Benchmarks

The benchmarks are included in the artifact. For path resolution, we consider two categories of scenarios: file type identification and path normalization. The artifact includes the symlink path cases used to support our experimental analysis. During execution, the artifact dynamically generates the required directory structures for these cases. In addition, we provide testing wrappers for 35 file type identification interfaces and 34 path normalization interfaces across ten programming languages. For path validation, the artifact includes wrappers and dependency scripts for 85 archive extraction projects, together with scripts that dynamically generate the test archives used in our experiment.

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

Download and extract the archived artifact, then enter the repository root:

```
cd SymlinkMisunderstanding
```

Build the path-resolution image:

```
cd PathResolution
docker build -t symlink-pathresolution .
```

Build the path-traversal image:

```
cd ../PathTraversalFinding
docker build -t ptr-check .
```

A.3.2 Basic Test

After building the Docker image, run a basic functionality test:

```
docker run --rm symlink-pathresolution bash -c "echo OK"
docker run --rm ptr-check bash -c "echo OK"
```

Expected output: `OK`. This confirms that the Docker image is correctly built and the language runtime environments are functional.

A.4 Evaluation workflow

This section includes the operational steps and experiments required to evaluate the functionality of our artifact and validate the key results and claims of our paper.

A.4.1 Major Claims

Our major claims include the following:

(C1): *Path resolution behavior experiments.* We identify four aspects of confusion in file type identification and five distinct normalization strategies across ten languages.

(C2): *Path traversal vulnerability detection.* Our tool can automatically detect path traversal vulnerabilities in archive extraction projects. Through evaluation of 85 projects, we uncover 16 arbitrary file write vulnerabilities.

A.4.2 Experiments

The corresponding experimental steps are as follows:

(E1): *Path resolution behavior experiments [30 human-minutes + 1 compute-hours]*
Reproduce the path resolution inconsistency analysis across 10 programming languages.
Preparation: Enter `PathResolution/` directory. Mount `PathResolution` using an absolute path:

```
docker run -it \
    -v "$PWD:/workspace/PathResolution" \
    symlink-pathresolution bash
```

Execution: Inside the container, run the file type identification tests:

```
cd /workspace/PathResolution/FileTypeIdentification
./run/run_all.sh
```

Then run the path normalization tests:

```
cd /workspace/PathResolution/PathNormalization
./run_all.sh
python3 classify.py
```

Results: CSV files are generated under `FileTypeIdentification/Results/` and `PathNormalization/Results/`, with one file per language. The classification script outputs a strategy table (S1–S4, Abs-Rel) and writes `Results/_strategy_classification.csv`. Results should match the pre-computed outputs in `ExperimentResults/`.

(E2): *Path traversal vulnerability detection [30 human-minutes + 2 compute-hours]*
Reproduce the path traversal vulnerability detection on 85 archive extraction projects.
Preparation: Enter `PathTraversalFinding/` directory. Mount PathTraversalCheck using an absolute path:

```
docker run -it -v \
 "$PWD/PathTraversalCheck:/opt/ptr/PathTraversalCheck" \
 ptr-check
```

Prepare dataset dependencies inside the container:

```
cd /opt/ptr/PathTraversalCheck
bash datasets/setup_all.sh
```

Set up the Python environment:

```
cd /opt/ptr/PathTraversalCheck
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Execution: Run the detection tool:

```
python main.py
```

Results: The tool outputs a summary of vulnerability detection results. The expected output should match:

```
summary:
  total: 85
  arbitrary write: 16
  bypass arbitrary read: 2
  arbitrary read: 22
  no issue: 45
```

Detailed per-project results are written to `PathTraversalCheck/out/`, including generated PoC archive files under the `seeds/` subdirectories.

A.5 Notes on Reusability

The artifact can be extended to test additional archive extraction libraries. To add a new target, users can add an entry to `test.json` and provide the corresponding wrapper implementation. The bypass and write strategies implemented in the tool are modular and can be adapted to evaluate other path validation scenarios beyond archive extraction.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: PROBE+DETECT+MITIGATE (PDM): Enabling Cloud Tenants to Self-Defend against Microarchitectural Attacks

Arash Daneshmand[2,1✉], Hugo Kermabon-Bobinnec[1],
Lingyu Wang[2,1✉], Makan Pourzandi[3], Suryadipta Majumdar[1], and Yosr Jarraya[3]

[1]CIISE, Concordia University, Montreal, QC, Canada
[2]School of Engineering, University of British Columbia, Kelowna, BC, Canada
[3]Ericsson Security Research, Ericsson Canada, Montreal, QC, Canada
{hugo.kermabonbobinnec,arash.daneshmand, suryadipta.majumdar}@concordia.ca
lingyu.wang@ubc.ca
{makan.pourzandi,yosr.jarraya}@ericsson.com

A Artifact Appendix

A.1 Abstract

This artifact demonstrates the functionality of the proposed detection and mitigation mechanisms as mentioned in the Open Science Section of our paper. It confirms that the provided artifacts correspond to the system design presented in the paper. The artifact includes the source code of PDM's detection and mitigation components, including probing schemes, runtime ML-based inference using ONNX, and in-memory encryption. It also includes the trained models and datasets used in our experiments, which cover the normal activities of real-world applications, attack activities, natural noises from SPEC CPU 2017, artificial noises from memory stressors, and slow evasive attacks. The complete compilation and usage instructions are included in this appendix and also in the README files provided.

A.2 Description & Requirements

This section lists all the necessary details to recreate the same experimental setup we have used to test the artifact.

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, and ethical concerns for evaluators while running our artifact on their machine. No destructive steps such as disabling security features are taken in the process. The artifact uses standard user-space techniques and evaluators should run it only on processes they own.

A.2.2 How to access

Our artifact is accessible through Zenodo at https://doi.org/10.5281/zenodo.17967755, and additionally maintained in Github at https://github.com/arashd021/pdm.

A.2.3 Hardware dependencies

We implemented and evaluated our artifact using (i) an in-house testbed with an Intel Xeon Gold 5120 (Skylake) CPU with 19.25MB L3 cache, and 1MB L2 cache per core, and (ii) a Kubernetes cluster deployed on Amazon EKS based on Amazon Fargate with one vCPU and 1GB of memory. The artifact requires an x86-64 machine with a minimum of one hardware thread (as PDM does not require any additional thread).

A.2.4 Software dependencies

We tested our artifact on a fresh Ubuntu 24.04 installation as of December 2025 but it should run on any x86-64 Linux operating system. The following software dependencies are required to run the artifact:

- Python (detection datasets and models were tested with Python 3.9+)
- Python packages (pip): pandas, numpy, torch, scikit-learn, joblib (to test the detection datasets and models)
- Jupyter (to run the .ipynb notebooks)
- System build tools: gcc / g++ / make / util-linux
- libcapstone-dev (required by mitigation)
- uthash (required by mitigation)

- libssl-dev (required by OpenSSL-based victim examples for mitigation)
- libwolfssl-dev (required by wolfSSL victim examples for mitigation)
- ONNX Runtime (to enable low-overhead runtime ML inference in C)
- The CAP_SYS_PTRACE Linux capability (for runtime thread injection).
- Docker (to run the provided container for mitigation)

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Install the required software dependencies:

```
$ sudo apt update && sudo apt install -y
build-essential libcapstone-dev libssl-dev
libwolfssl-dev uthash-dev python3-pip
```

Install ONNX Runtime (v1.22.0) and set `ORT_DIR` (expected by the detection build):

```
$ cd ~
$ wget https://github.com/microsoft/
onnxruntime/releases/download/v1.22.0/
onnxruntime-linux-x64-1.22.0.tgz
$ tar -xvf onnxruntime-linux-x64-1.22.0.tgz
$ export ORT_DIR=$HOME/onnxruntime-linux-x64-1.2
2.0
```

Download the latest release of the artifact from Zenodo and extract it:

```
$ cd ~
$ wget <Zenodo Download URL>
$ unzip <PDM Zipped File>
$ cd <PDM Directory>
```

Install Python dependencies:

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

Build the artifact:

```
$ cd ../source/detection && make
$ cd ../mitigation && make
```

PDM's probing requires three distinct thresholds for different cache levels. Since these thresholds are different across environments, we provide a calibration tool to automatically identify these thresholds. Simply run the tool:

```
$ cd source/detection
$ ./calibration
```

You should then set the calibrated thresholds by updating the threshold constants (`THR_L1`, `THR_L3`, `THR_MISS`) at the top of *PDM-detection.c*, *PDM-scheme3.c*, and *PDM-scheme3+.c*, and recompile the detection components.

Finally, install the runtime `.so` library injection tool:

```
$ git clone https://github.com/kubo/injector
$ cd injector && make
```

A.3.2 Basic Test

Detection. The `example/fr` directory contains an example victim application, a FLUSH+RELOAD attack, and integration with PDM for detection:

```
$ cd source/detection/example/fr
$ make
```

The flag `#define USE_PROC_MAPS` in PDM's probing codes (enabled by default), allows the probing range to be automatically identified. Run the victim binary with PDM integrated:

```
$ sudo LD_PRELOAD=../../PDM-detection.so taskset
-c 0 ./victim
```

You should see benign datapoints where the L1hit ratio is 100% and the model output is 0:

```
# 14:55:43.967 l1hit: 100.00 ** l3hit: 0.00 **
miss: 0.00 ** bigmiss: 0.00 ===== l1hit2: 0.00
** l3hit2: 0.00 ** miss2: 100.00 ** bigmiss2:
0.00 | Model Output: 0.000000 ** batch_offset:
2560
```

Next, in another terminal, run the attack

```
$ ./F+R
```

You should see a decrease in L1 hit ratio and an increase in model output:

```
# 14:57:37.019 l1hit: 25.00 ** l3hit: 0.00 **
miss: 62.50 ** bigmiss: 12.50 ===== l1hit2: 0.00
** l3hit2: 0.00 ** miss2: 100.00 ** bigmiss2:
0.00 | Model Output: 0.800035 ** batch_offset: 0
```

Mitigation. A ready-to-use Docker image with precompiled ELF 64-bit binaries of different cryptographic primitives with and without the in-memory encryption mitigation is provided. Run the container interactively:

```
$ docker run --rm -it docker.io/pdmanon123/
pdm-artifacts
```

You can then run each binary in two modes (with and without protection). For instance:

```
$ ./ecdh 1 #without protection
$ ./ecdh 2 #with protection
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PDM *detects cache-based side-channel attacks (e.g., FLUSH+RELOAD) using only tenant-accessible features (i.e., user-space probing and ONNX-based inference), without requiring cloud-provider support. This is validated by experiment (E1).*

(C2): PDM *detects attacks relying on speculative memory accesses (e.g., Spectre variants) using the same tenant-side workflow and features, generalizing beyond eviction-based cache attacks. This is validated by experiment (E2).*

(C3): PDM *protects secrets via the paper's in-memory protection mechanism (instruction self-declaration and on-the-fly inline binary rewriting), preserving application correctness across multiple crypto libraries and primitives. This is validated by experiment (E3).*

(C4): PDM *remains effective under realistic noise and evasion conditions, including (i) natural co-resident noise (SPEC CPU 2017) on a local testbed, (ii) artificial memory stressors on a local testbed, (iii) real-cloud noise on AWS Fargate, and (iv) slow/evasive attacks. This is validated by experiments (E4).*

A.4.2 Experiments

(E1): *[FLUSH+RELOAD detection on OpenSSL AES]* [30 human-minutes + 20 compute-minute] This experiment reproduces a FLUSH+RELOAD attack against OpenSSL AES-128 (T-table implementation) and demonstrates that PDM's detection produces clearly separable outputs between benign execution and attack execution.

Preparation: This experiment requires a self-compiled version of OpenSSL to enable its T-Table-based AES implementation. Build and install a vulnerable OpenSSL with T-table AES enabled (OpenSSL 1.1.0f) under `/usr/local/ssl_vuln`, then compile the provided attacker/victim programs (this will not replace your currently installed OpenSSL).

```
$ cd source/detection/example/fr-aes
$ wget https://www.openssl.org/source/
openssl-1.1.0f.tar.gz
$ tar -xvf openssl-1.1.0f.tar.gz
$ cd openssl-1.1.0f
$ ./config -d shared no-asm no-hw --prefix=/
usr/local/ssl_vuln
$ sudo make && sudo make install_sw
```

Ensure runtime can find the vulnerable libcrypto:

```
$ export LD_LIBRARY_PATH=/usr/local/ssl_vuln
/lib:$LD_LIBRARY_PATH
```

Compile the applications:

```
$ cd ..
$ g++ spy.cpp -o spy -I/usr/local/ssl_vuln/i
nclude -L/usr/local/ssl_vuln/lib/ -lcrypto
$ gcc aes-workload.c -o aes-workload -I/usr/
local/ssl_vuln/include -L/usr/local/ssl_vul
n/lib/ -lcrypto
```

PDM can be configured to use fixed address ranges (e.g., AES T-tables addresses) in `source/detection/PDM-detection.c`. T-table addresses can simply be found by running the binary (keep the process running so addresses do not change across runs due to ASLR) :

```
$ ./spy
Probing address (Te0): 0x7ffff6fdfc00
Probing address (Te1): 0x7ffff6fdf800
Probing address (Te2): 0x7ffff6fdf400
Probing address (Te3): 0x7ffff6fdf000
```

Verify that the following macros are set using the address of `Te3` and recompile PDM:

```
#define USE_FIXED_START_ADDR
#define START_ADDR 0x7ffff6fdf000
#define SIZE 3072
```

Execution: Run the spy process, then start the AES workload, inject `PDM-detection.so` into the spy process, and toggle the attack on/off using signals.

```
# Terminal 1: spy already running (printed
probed Te* addresses)
```

```
# Terminal 2: start continuous AES
encryptions (victim workload)
$ ./aes-workload 1
```

```
# Terminal 3: inject PDM detection into the
spy process
```

```
$ cd source/detection/injector
$ sudo ./cmd/injector -p $(pidof spy)
PDM-detection.so
# Toggle attack (in the spy process)
# Enable attack
$ kill -SIGUSR1 $(pidof spy)
# Disable attack
$ kill -SIGUSR2 $(pidof spy)
```

Results: PDM prints the multivariate time-series and the ONNX model output (in Terminal 1). When the attack is enabled (`SIGUSR1`), the model output should shift compared to the benign phase (`SIGUSR2`), matching the eviction effect reported in the paper.

(E2): *[Spectre detection]* [20 human-minutes + 10 compute-minute] This experiment shows that PDM can detect *Spectre-PHT*, as an example of attacks relying on speculative memory accesses.
Preparation: Build the modified Spectre-PHT proof-of-concept.

```
$ cd source/detection/example/spectre-pht
$ make
```

Similar to the previous experiment, PDM uses fixed-address mode for this experiment. First, run the Spectre PoC to extract the address of secret:

```
$ ./spectre.out
# The virtual address of 'secret' is:
0x555555556008
# Length of the string in secret: 40
characters
```

Next, verify that the following macros are enabled in `source/detection/PDM-detection.c` with printed address of the secret, and recompile PDM.

```
#define USE_FIXED_START_ADDR
#define START_ADDR 0x555555556008
#define SIZE 40
```

Execution: Run the PoC:

```
$ taskset -c 0 ./spectre.out
```

In a second terminal, inject PDM to the PoC process:

```
$ cd source/detection/injector
$ sudo ./cmd/injector -p $(pidof
spectre.out) PDM-detection.so
```

And toggle the attack on/off:

```
# Enable attack
$ kill -SIGUSR1 $(pidof spectre.out)
# Disable attack
$ kill -SIGUSR2 $(pidof spectre.out)
```

Results: When the attack is enabled, the PDM probing output should show a noticeable increase in the `L1hit2` ratio, and the printed ONNX inference score should shift consistently relative to the benign phase.

(E3): *[In-memory Encryption]* [30 human-minutes + 10 compute-minute]
This experiment demonstrates PDM's in-memory encryption mitigation. After guarding secret pages, accesses trigger a signal handler that performs on-the-fly inline binary rewriting so that the application encrypts/decrypts secrets transparently through injected trampolines. The protected application preserves its functionality correctness and terminates normally across different crypto primitives.
Preparation: The mitigation signal handler and example crypto primitives are already compiled as mentioned in Section A.3.1:
Execution: Run each crypto primitive binary in two modes with the handler injected using LD_PRELOAD. For instance:

```
$ cd source/mitigation/example/openssl
$ LD_PRELOAD=../../PDM-encrypt.so ./ecdh 1
$ LD_PRELOAD=../../PDM-encrypt.so ./ecdh 2
```

Results: Each mode prints the stored cryptographic key in the memory and the execution time. Mode 1 runs normally using the actual key stored in memory, without protection. Mode 2 should print that the key is being guarded, show its encrypted content in memory, and still complete successfully.

(E4): *[Datasets and Models]* [30 human-minutes + 10 compute-minute] This experiment shows PDM's robustness by running the provided Jupyter notebooks on the different datasets (testbed, AWS Fargate, and noise). The notebooks load the corresponding trained model (`.pth`/`.pkl`), run inference on the selected dataset (`.csv`), and report metrics (e.g., TPR/FPR).
Preparation: Required dependencies for this experiment are mentioned in Section A.3.1. Run Jupyter Notebook:

```
$ cd datasets
$ jupyter notebook
```

Execution: Open any of the following notebooks and run the cells. Each notebook contains a variable `dataset_path` that selects which `.csv` file to evaluate.

- `datasets/testbed/Detection Engine (In-`

ference Only).ipynb (local testbed datasets, including FLUSH+RELOAD, FLUSH+FLUSH, PRIME+PROBE and Spectre variants and co-resident natural noises from SPEC CPU 2017 benchmark).

- datasets/noise/Detection Engine 8 (Inference Only).ipynb and datasets/noise/Detection Engine 9 (Inference Only).ipynb (memory stressor noise).
- datasets/fargate/Detection Engine (Inference Only).ipynb (real-cloud AWS Fargate datasets, including FLUSH+RELOAD, FLUSH+FLUSH and Spectre variants).

Results: For each run, the notebook prints detection metrics (TPR/FPR) and saves an output file named inference_output_with_metrics.csv in the same directory showing the model's output for each datapoint.

A.5 Notes on Reusability

PDM's detection and mitigation components are both intentionally modular. They are both compiled as user-space shared libraries that can be loaded into the application via LD_PRELOAD or ptrace runtime injection. To apply PDM's detection to a new application, evaluators can (i) identify the secret addresses from the running application and update detection's probing range, (ii) run the included cache threshold calibration tool on the target machine and update the probing thresholds accordingly, and (iii) compile the probing/detection shared library and load it into the target process using the provided injector. The mitigation handler can be similarly injected to a target application. The provided examples demonstrate the expected integration pattern and can be used as templates for additional crypto libraries or primitives.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Charting the Cache Side-Channel Frontier: A Systematic Study of Eviction Set Construction on Apple Silicon

Han Wang[†] Yakun Wu[†] Yusi Feng[†] Yinqian Zhang[†‡]
Southern University of Science and Technology

A Artifact Appendix

A.1 Abstract

This paper presents a systematic study of eviction-set construction on Apple Silicon, focusing on the Apple M4 microarchitecture. Eviction sets are a fundamental primitive for conflict-based cache side-channel attacks, but constructing reliable eviction sets on recent Apple Silicon is challenging due to undocumented microarchitectural behavior. Our work reverse engineers the root causes of these challenges, including an SLC victim buffer that obscures latency-based verification, a DRAM-to-Cache Prefetcher that disrupts strict LRU assumptions, and an opaque SLC allocation policy influenced by L2 insertion and promotion behavior.

The artifact is designed to support the main experimental claims of the paper. It includes benchmark programs, reverse-engineering experiments, eviction-set construction algorithms, data-processing scripts, and plotting scripts. Through the artifact, evaluators can reproduce the characterization of key Apple M4 cache behaviors, validate the effectiveness of our optimized L2 eviction-set construction strategy, and evaluate the hierarchical method used to construct fine-grained SLC eviction sets.

Because the results depend on Apple M4-specific microarchitectural behavior, full artifact evaluation requires evaluator-side access to a local Apple M4 machine. In the absence of such hardware, the artifact remains useful for reviewing the implementation and experimental workflow, but the main timing measurements and eviction-set construction results cannot be fully reproduced. On the target platform, the artifact provides the necessary code and scripts to reproduce the paper's core cache-behavior observations, timing measurements, and construction results. Exact timing distributions and success rates may vary slightly depending on system noise, OS configuration, core placement, and runtime conditions.

[‡]Corresponding author.
[†]Affiliated with the Research Institute of Trustworthy Autonomous Systems and the Department of Computer Science and Engineering.

A.2 Description & Requirements

This artifact accompanies *Charting the Cache Side-Channel Frontier: A Systematic Study of Eviction Set Construction on Apple Silicon*. It contains the source code, benchmark programs, eviction-set construction algorithms, reverse-engineering experiments, data-processing scripts, and plotting scripts used to reproduce the main experimental results in the paper. The artifact is organized by experiment modules, covering cache-latency characterization, DRAM-to-Cache Prefetcher (DCP) analysis, L2 Insertion/Promotion Vector (IPV) and SLC allocation characterization, optimized L2/SLC eviction-set construction, and downstream security-evaluation experiments.

The main experimental setup used in the paper is an Apple M4 Mac mini. The experiments rely on high-resolution cycle measurements through the Apple PMU register `S3_2_c15_c0_0` and on data-cache maintenance instructions such as `dc civac`. Therefore, evaluators should prepare a dedicated Apple Silicon machine and enable these instructions following the PacmanPatcher setup described below.

A.2.1 Security, privacy, and ethical concerns

This artifact is security-sensitive and dual-use. It studies cache side channels and eviction-set construction on Apple Silicon. The released artifact is scoped to reproduce the paper's scientific claims, including microarchitectural measurements and eviction-set construction, and is not intended to provide a weaponized exploit or a turnkey exploitation pipeline.

Running the artifact may require patching the macOS kernel to enable user-space access to performance counters and cache-maintenance instructions. This can reduce system security, may require changes to boot/security settings, and can cause system instability, crashes, or boot failures. Evaluators should run the artifact only on a dedicated, non-production machine that they own or are explicitly authorized to use. The machine should not contain sensitive user data, and evaluators should keep a recovery plan and backup before applying any kernel patch.

The experiments operate on locally allocated memory buffers and do not require third-party accounts, network ser-

vices, personal data, or access to external systems. Evaluators should not run the artifact on shared, production, or third-party systems and should avoid collecting data from non-consenting users or workloads.

A.2.2 How to access

A stable archived snapshot of the artifact is available at:

https://doi.org/10.5281/zenodo.20336892

A.2.3 Hardware dependencies

The main results require an Apple M4 Mac mini, matching the experimental setup used in the paper. The relevant microarchitectural features are the Apple M4 CPU cache hierarchy, including private L1 caches, the shared L2 cache within a CPU cluster, and the system-level cache (SLC). The paper's measurements assume 128-byte cache lines, a 16 MiB L2 cache, and an 8 MiB SLC on the evaluated M4 platform.

A machine with at least 16 GiB of RAM is recommended, because some L2 candidate-pool experiments sweep candidate pools up to 1 GiB. No GPU, FPGA, or special accelerator is required. Physical or administrative access to the machine is required because the setup may involve installing a patched kernel collection and booting through macOS recovery/1TR mode.

A.2.4 Software dependencies

Please set up the environment by following the instructions in the PacmanPatcher code repository: https://github.com/jprx/PacmanPatcher.

In addition, the artifact requires the following software environment:

- macOS on Apple Silicon.
- The Apple Kernel Debug Kit (KDK) matching the evaluator's macOS build, as required by PacmanPatcher.
- A C/assembly build environment for ARM64/AArch64 code.
- Python 3 for data processing and plotting.
- Python packages required by the plotting and analysis scripts, such as `numpy`, `pandas`, `scipy`, and `matplotlib`.

A.2.5 Benchmarks

None.

A.3 Set-up

This section describes the installation and configuration steps needed to prepare the evaluation environment.

A.3.1 Installation

Set up PacmanPatcher by following the instructions in: https://github.com/jprx/PacmanPatcher. In particular, download the KDK matching the macOS build, build PacmanPatcher, patch the kernel as instructed, create a patched kernel collection, and boot the machine using the patched kernel collection. After booting into the patched kernel, verify that PMU access and cache-maintenance instructions work:

```
cd <PacmanPatcher-root>
./test_pmc
./test_flush
```

Both tests should complete without illegal-instruction failures or crashes. The PMU test should show that cycle-counter reads are available from user space, and the flush test should show that cache-maintenance instructions are executable from user space.

Each experiment module should be built according to its local README or Makefile. Build the relevant module before running it, for example:

```
cd <artifact-root>/01_Benchmark/01_cache_latency
make
```

A.3.2 Basic Test

A minimal functionality test should verify three things: PMU access, cache flush support, and the ability to compile and execute one benchmark.

Step 1: Run the PacmanPatcher checks:

```
cd <PacmanPatcher-root>
./test_pmc
./test_flush
```

Expected output: The tests should terminate normally. The PMU test should report usable cycle-counter behavior, and the cache-flush test should confirm that cache-maintenance instructions are available from user space.

Step 2: Build and run a small cache-latency benchmark:

```
cd <artifact-root>/01_Benchmark/01_cache_latency
make
python3 run_test_dcache_single_avarage.py
python3 plot_test_dcache_single.py
```

Expected output: The benchmark should produce timing samples or an intermediate result file. The corresponding analysis/plotting script should classify accesses into latency regions consistent with the paper: L1/L2 hits below roughly 130 cycles, SLC hits around 170–300 cycles, and DRAM accesses around 400–600 cycles.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): C2SCF characterizes the Apple M4 cache hierarchy and identifies latency regions corresponding to L1/L2

hits, SLC hits, SLC victim-buffer accesses, and DRAM accesses. It further shows that random access leads to low effective SLC utilization and that the SLC victim buffer masks eviction events. This is proven by the experiments in Sec. 3, whose results are shown in Fig. 1 and Fig. 2.

(C2): C2SCF uncovers a DRAM-to-Cache Prefetcher (DCP) on Apple M4. The DCP can reload flushed data into the private cache hierarchy, can be triggered by a small set of precursor accesses, follows constrained spatial patterns, and can interfere with Flush+Reload-style decoders. This is proven by the experiments in Sec. 4, whose results are shown in Fig. 3–Fig. 6 and discussed in Sec. 4.5.

(C3): C2SCF shows that L2 replacement behavior and SLC allocation are coupled through L2 Insertion/Promotion Vectors (IPVs). Random access patterns receive higher residency priority than strided patterns, repeated evictions can elevate cacheline priority, and high-IPV evictions tend to trigger rapid propagation toward DRAM rather than stable SLC residency. This is proven by the PIP and I2P experiments in Sec. 5, whose results are shown in Fig. 8, Fig. 10, Fig. 11, and Table 1.

(C4): C2SCF's Apple-specific L2 construction optimizations substantially improve L2 eviction-set construction. Generic PPP fails under Apple M4 noise, while optimized PPP reaches the reported L2 eviction-set construction success rate of 88.2%. This is proven by the experiments in Sec. 6.1–Sec. 6.2, whose results are shown in Fig. 12, Fig. 13, Table 2, Table 3, and Table 4.

(C5): C2SCF constructs the first practical fine-grained SLC eviction sets on Apple Silicon using hierarchical filtering. The optimal SLC construction reaches the reported success rate of 90%, and the resulting primitives provide useful temporal/spatial resolution for downstream side-channel evaluation. This is proven by the experiments in Sec. 6.3–Sec. 6.5, whose results are shown in Fig. 14–Fig. 17.

A.4.2 Experiments

For each experiment, the exact executable names, command-line arguments, output files, and plotting commands are provided in the corresponding module-local README.

(E1): Cache hierarchy and SLC victim-buffer characterization. This experiment supports claim C1.

How to: Use the benchmark modules `01_Benchmark/01_cache_latency`, `01_Benchmark/02_cache_flush_latency`, and `01_Benchmark/03_slc_victim_buffer`.

Preparation: Build the C benchmark programs in each directory according to the module-local README or Makefile. Ensure that PMU timing and cache flushing work by running `test_pmc` and `test_flush` from the PacmanPatcher setup.

Execution: Run the dense random-access benchmark, the flush-after-random-access benchmark, and the sparse conflict-access benchmark. The sparse conflict-access benchmark should use a page-size interval to induce high contention and expose the SLC victim-buffer behavior.

Results: Process the generated timing data with the corresponding Python scripts. The expected outcome is the reproduction of Fig. 1a, Fig. 1b, Fig. 1c, Fig. 2a, and Fig. 2b. The results should show distinct latency regions for L1/L2, SLC, and DRAM, low effective SLC utilization under random access, anomalous low-latency hits after flushing, and a split in the SLC-latency range consistent with an SLC victim buffer.

(E2): DCP behavior, triggering pattern, mitigation, and case study. This experiment supports claim C2.

How to: Use the modules `02_DCP/01_behavior`, `02_DCP/02_minimal`, `02_DCP/03_pattern`, `02_DCP/04_mitigation`, and `02_DCP/05_case`.

Preparation: Build the benchmark programs and activate the Python environment.

Execution: First, run the DCP behavior experiment using pseudo-random linear-shift access and fully random access patterns. Next, run the reduction-based minimal-trigger experiment to identify precursor cachelines. Then run the spatial-pattern enumeration experiment across different address intervals. Finally, run the mitigation experiment with temporal delays and address offsets, and run the Flush+Reload decoder case study.

Results: Process and plot the results to reproduce Fig. 3, Fig. 4, Fig. 5, and Fig. 6, and to reproduce the qualitative result discussed in Sec. 4.5. The expected outcome is that forced flushing does not always lead to DRAM-latency reloads, because the DCP can bring data back into the cache hierarchy. The minimal-trigger results should show that a small precursor sequence can activate the DCP. The spatial-pattern results should show constrained power-of-two-style address deltas. The mitigation results should show that large address intervals and temporal perturbation reduce DCP interference.

(E3): L2 IPV dynamics and SLC allocation policy. This experiment supports claim C3.

How to: Use the modules `03_IPV/01_PIP`, `03_IPV/02_I2P/01_heatmap`, and `03_IPV/02_I2P/02_transition`.

Preparation: Build the PIP and I2P benchmark programs.

Execution: Run the PIP experiment groups that vary prime, interfere, and probe patterns. Then run the I2P experiment over multiple rounds and working-set sizes to track how lines move through L2, SLC, the victim buffer, and DRAM.

Results: Generate the plots and tables corresponding to Fig. 8, Fig. 10, Fig. 11, and Table 1. The expected outcome is that random priming yields stronger L2 resi-

dency than strided priming, strided interference is less effective at evicting high-priority resident lines, repeated access/eviction cycles change retention behavior, and SLC allocation depends on the L2 state.

(E4): Optimized L2 eviction-set construction. This experiment supports claim C4.

How to: Use the modules `04_EV/01_L2/01_access_frequency`, `04_EV/01_L2/02_backtrack`, `04_EV/01_L2/03_candidates`, and `04_EV/01_L2/04_random`.

Preparation: Build the L2 eviction-set construction programs.

Execution: Run the baseline and optimized L2 construction algorithms, including SHM, GE, CT, and PPP variants. Evaluate the effects of access randomization, access frequency, backtracking or majority-voting robustness, and candidate-pool size.

Results: Reproduce Fig. 12, Fig. 13, Table 2, Table 3, and Table 4. The expected outcome is that generic PPP performs poorly on Apple M4 unless the Apple-specific optimizations are applied. With the optimized parameters, PPP should reproduce the reported high L2 eviction-set construction success rate, while large candidate pools should demonstrate the degradation caused by DCP interference.

(E5): Hierarchical SLC eviction-set construction and downstream security evaluation. This experiment supports claim C5.

How to: Use the modules `04_EV/02_SLC` and `04_EV/04_sec_eva`.

Preparation: First construct valid L2 eviction sets, because the SLC construction uses L2 filters as part of the hierarchical filtering process. Configure the number of L2 filters and select the construction strategy to evaluate, such as GE+GE, PPP+GE, or PPP+PPP.

Execution: Run the hierarchical SLC construction experiment over the configured number of L2 filters. Then run the stability evaluation by repeatedly applying the eviction sequence to successful eviction sets. Finally, run the temporal-resolution, spatial-resolution, and AES T-table cache-line classification experiments.

Results: Reproduce Fig. 14–Fig. 17 and the results discussed in Sec. 6.5. The expected outcome is that SLC construction success improves as the number of L2 filters increases, the best SLC construction reproduces the reported 90% success rate, and the optimized L2 eviction sets provide clear cache-line-level spatial resolution. The downstream AES T-table benchmark should show substantially better classification accuracy for the optimized construction than for the minimally adapted PPP baseline.

A.5 Notes on Reusability

The artifact is organized so that individual modules can be reused independently. For example, the `01_Benchmark` modules can be reused to calibrate cache latency thresholds on a new Apple Silicon machine, the `02_DCP` modules can be reused to study prefetcher behavior, and the `03_IPV` modules can be reused to characterize replacement and allocation policies.

When porting the artifact to another Apple M-series chip or another ARM64 platform, users should first rerun the cache-latency calibration benchmarks and update the latency thresholds, cache sizes, cache-line assumptions, and candidate-pool parameters accordingly. The eviction-set construction modules should then be rerun with the calibrated parameters. Results may differ across SoCs and macOS versions because the relevant prefetchers, replacement policies, PMU access mechanisms, and cache-maintenance behavior are microarchitecture- and OS-dependent.

For safe reuse, the artifact should remain limited to controlled local experiments on owned or authorized hardware. Users should avoid integrating the artifact into automated exploit pipelines or running it against third-party systems or sensitive workloads.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: microSCALE: Static Analysis for Microarchitectural Side-channel Leakage Evaluation

Akshay Kumar E*
IIT Madras, India

Pranav Krishna N*
IIT Madras, India

Annapurna Valiveti
IIT Madras, India

Pallavi Borkar
IIT Madras, India

Aditi Roy
IIT Madras, India

Chester Rebeiro
IIT Madras, India

A Artifact Appendix

A.1 Abstract

The artifact for microSCALE is a static analysis framework for detecting power side-channel leakages in masked cryptographic implementations running on a target processor. The framework statically analyzes the processor's RTL design to detect and localize transitional leakages at both the hardware and software levels. The artifact is packaged as a Docker image that encapsulates all dependencies, and includes the microSCALE source code along with pre-built opcode-specific subgraphs for the RISC-V Rocket Core.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

There are no security, privacy, or ethical risks to evaluators. microSCALE is a purely defensive static analysis tool. No system security mechanisms are disabled, no destructive operations are performed, no sensitive data is used, and no network access is required during execution. All analysis runs entirely within the provided Docker container.

A.2.2 How to access

The artifact is publicly available at:

https://zenodo.org/records/20409592

The repository contains the microSCALE source code, pre-built processor graphs for the Rocket Core, cipher assembly programs used in the evaluation, and a Dockerfile for reproducible execution.

A.2.3 Hardware dependencies

None. microSCALE requires only a standard x86-64 machine capable of running Docker.

A.2.4 Software dependencies

All software dependencies are encapsulated in the provided Docker image, ensuring a consistent and portable execution environment. The only requirement on the host machine is **Docker** (any recent version supporting `docker build` and `docker run`). Internally, the container runs Python 3.10. The following Python scripts implement the framework, and the Dockerfile provides easy, single-command access to the artifact.

`graph_maker.py` Constructs the Microarchitectural Dependency Graph (MDG) from the processor RTL.

`lookup_table_maker.py` Builds opcode-specific subgraphs from the MDG (pre-computed for RocketChip and included in the image).

`main.py` Main microSCALE pipeline: performs operand propagation and produces per-instruction data propagation maps.

`leakage_checker1.py` First-order leakage detection (default).

`leakage_checker2.py` Second-order leakage detection.

These scripts are not intended to be invoked directly. **All interaction with microSCALE should go through Docker** as described in Section A.3.

A.2.5 Benchmarks

The artifact includes the following assembly programs corresponding to the benchmarks in Table 4 of the paper: `gift_1round`, `giftsbox`, `AES_sbox`, `PRESENT_sbox`, `ASCON_sbox`, `isw_xor`, `dom_and`, `full_refresh`, and `locality_refresh`. tHE programs are included in the Docker image under `/app/programs/`.

A.3 Set-up

A.3.1 Installation

Download and extract the artifact from Zenodo, enter the directory, and build the Docker image:

```
docker build -t microscale .
```

This installs all dependencies inside the container. The RocketChip MDG and opcode-specific subgraphs are precomputed and bundled; no additional setup is required.

A.3.2 Basic Test

Run a small benchmark to verify the installation:

```
docker run microscale isw_xor
```

Expected output: a leakage report listing the leaking instruction pairs and responsible microarchitectural registers, consistent with Table 4 of the paper. The following predefined programs are also available:

```
docker run microscale dom_and
docker run microscale full_refresh
docker run microscale locality_refresh
docker run microscale giftsbox
docker run microscale AES_sbox
docker run microscale PRESENT_sbox
docker run microscale ASCON_sbox
docker run microscale gift_1round
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): microSCALE detects transitional leakage in masked cryptographic implementations running on the Rocket Core, producing a leakage report that identifies leaking instruction pairs and the responsible microarchitectural registers. This is demonstrated by experiment (E1) described in Section 6 of the paper, whose results are reported in Table 4.

A.4.2 Experiments

(E1): *Leakage detection on included benchmarks*: Run microSCALE on the included benchmark programs and verify that a leakage report is produced for each, consistent with Table 4 of the paper.
How to: Run each predefined benchmark using Docker and inspect the leakage report printed to stdout.
Preparation: Complete the Docker build described in Section A.3. No additional configuration is needed.
Execution: Run, for example, the following commands:
`docker run microscale isw_xor`
`docker run microscale AES_sbox`
For testing with a custom RISC-V assembly file, see the example in Section A.5.
Results: Each run produces a transitional leakage report.

A.5 Notes on Reusability

microSCALE accepts RISC-V assembly files. Custom masked implementations, written in the same RISC-V assembly style as the programs included under `/app/programs/`, can be analyzed by following the same workflow. To analyze a custom masked implementation, mount the file into the container and use the `custom` mode:

```
docker run \
  -v /path/to/your_file.s:/app/programs/your_file.s \
  microscale custom \
  --shares 2 --variables 2 --randoms 1 \
  --program programs/your_file.s \
  --graph Rocket --order 1
```

The `-shares`, `-variables`, and `-randoms` arguments specify the masking parameters of the input program.

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/usenixsec2026/.







USENIX Security '26 Artifact Appendix: Spectre on RISC-V Silicon: Attacks and Defenses on Commercial Out-of-Order Processors

Lukas Gerlach[1] Marton Bognar[2] Daniel Weber[1] Michael Schwarz[1] Jo Van Bulck[2]
[1]*CISPA Helmholtz Center for Information Security* [2]*DistriNet, KU Leuven*

A Artifact Appendix

A.1 Abstract

This artifact supports the Open Science scope stated in the paper. It contains the unified Spectre proof-of-concept framework, microarchitectural measurement tools, speculation-barrier tests, Smatch/CodeQL gadget-scanning scripts, cross-architecture mitigation-diffing code, branchless uBPF dispatch prototype and benchmark, optional SPEC CPU 2017 build setup, cache/timer primitives, BPF gadget-emitter source, and Linux kernel patches.

A.2 Description & Requirements

Full reproduction requires the evaluated RISC-V hardware.

A.2.1 Security, privacy, and ethical concerns

Several experiments show Spectre-style leakage, load kernel modules and change BPF settings. Run them only on isolated systems owned or explicitly authorized by the evaluator. The artifact does not process personal data or contact third-party systems.

A.2.2 How to access

Download the latest Zenodo release at `https://doi.org/10.5281/zenodo.18452278`. The relevant directories are:

- `spectre-poc-framework/`: 13 Spectre variants plus runner.
- `microarch-measurements/`: RSB-size and speculation-window tests.
- `barrier-test/`: candidate speculation-barrier tests.
- `gadet-scanning/`: Smatch and CodeQL kernel gadget scanning.
- `gadet-scanning/mitigation_diffing/`: semantic mitigation-diffing tool.
- `specbuild/`: SPEC CPU 2017 cross-build setup for mitigation benchmarks.
- `branchless-jit-dispatch/`: branchless dispatch prototype and modified uBPF.
- `cacheutils.h`, `counter-thread-timer/`: cache primitives and timer benchmark.
- `bpf-spectre-exploit/`: BPF Gadget Emitter: BPF bytecode to reliably emit the type confusion gadget and leak from a controlled kernel address.
- `kernel-patches/`: contributed RISC-V Spectre-v1 patches.

A.2.3 Hardware dependencies

The main hardware targets are T-Head Xuantie C910/C920 and SiFive P550 systems with root access. BTB experiments require BTB prediction to be enabled in firmware, which is not the case in the stock configuration of either board. P550 cache-eviction experiments require huge pages:

```
sudo sysctl -w vm.nr_hugepages=64
```

On some cores, userspace access to the cycle counter has to be enabled explicitly, otherwise the timed experiments abort with an illegal-instruction fault:

```
sudo sysctl -w kernel.perf_user_access=2
```

BPF experiments require a Linux target with BPF maps and JIT support.

A.2.4 Software dependencies

The target-board experiments (Basic Test, E1–E3, and E5) require a C/C++ toolchain, `make`, `cmake`, `bc`, Python 3, and the kernel/BPF settings listed per experiment. All experiments that are run on non RISC-V hardware (gadget scanning, cross-compiling SPEC CPU) are bundled in a Docker container that installs all the required dependencies.

A.2.5 Benchmarks

We use the `bpf-benchmark` to evaluate performance of our mitigated BPF interpreter. The proprietary SPEC CPU 2017 benchmark is optional. The `specbuild/` helper includes tooling to cross-compile the benchmarks on a fast machine and then run them on the target.

A.3 Set-up

A.3.1 Installation

Obtain the artifact from the provided Zenodo link and unpack it.

A.3.2 Basic Test

On a RISC-V target, run:

```
cd spectre-poc-framework
python3 run_all_tests.py --group baseline \
  --runs 1 --iterations 100
```

The runner should detect the platform, build the baseline binary, and print precision/recall metrics (compare to Table 2).

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) C910/C920 and P550 are vulnerable to major Spectre variants; reproduced by E1.

(C2) The BPF source emits the paper's speculative type-confusion gadget and shows that the leakage can be reproduced; reproduced by E2.

(C3) Speculation barriers, RSB size, speculation windows, cache channels, and counter-thread timing are processor-dependent; reproduced by E3.

(C4) Gadget scanning and mitigation diffing reveal missing RISC-V kernel hardening; reproduced by E4.

(C5) The branchless uBPF dispatch mitigation can be built and benchmarked with bundled BPF programs; reproduced by E5.

(C6, optional) The SPEC CPU 2017 mitigation setup reproduces the reported mitigation overheads given a licensed SPEC tree, a target sysroot, and dedicated RISC-V hardware; reproduced by optional E6.

A.4.2 Experiments

(E1) Spectre variant framework [15 human-minutes + 1-2 compute hours per board].

```
cd spectre-poc-framework
python3 run_all_tests.py --group all --runs 30 \
  --csv results.csv
```

Compare the CSV to Table 2 in the paper. The test runner also outputs human-readable logs with all the information. C910/C920 should show high recall for most PHT, RSB, and STL variants. P550 should show strong PHT sa-ip, RSB, and STL results with lower cross-address recall. The BTB variants only leak once BTB prediction is enabled in firmware, so they show no leakage on stock boards.

(E2) BPF proof-of-concept [20 human-minutes].

```
cd bpf-spectre-exploit
make clean && make exploit-complete
sudo sysctl -w kernel.perf_event_paranoid=0 \
  kernel.perf_user_access=2 \
  net.core.bpf_jit_enable=1 vm.nr_hugepages=1000
sudo ./exploit-complete
```

Expected result on the C910/C920: the program leaks the kernel banner. Additionally one can inspect the emitted RISC-V bytecode from the BPF JIT by setting:

```
echo 2 | sudo tee /proc/sys/net/core/bpf_jit_enable
```

This command dumps a lot of data to the kernel log.

(E3) Microarchitectural measurements [20 human-minutes + 1 compute-hour per board].

```
cd barrier-test && make
sudo ./barrier_test && sudo ./barrier_test_rsb
cd ../microarch-measurements/rsbsize-test
make && ./test
cd ../speculation-window
make && sudo ./main > spec-window.csv
cd ../../counter-thread-timer
sudo sysctl -w kernel.perf_event_paranoid=0 \
  kernel.perf_user_access=2
make && COUNTER_CORE=1 taskset -c 0 ./bench
```

Expected results: barrier tests report low hit rates for effective barriers. The RSB-size test shows a cliff near 12 entries on C910/C920 and 16 on P550. The counter-thread timer reports approximately cycle-scale resolution on C910/C920.

(E4, local) Kernel gadget scanning, mitigation diffing, and patches [1 human-hour + 1-2 compute-hours].

```
# Build the analysis image.
cd gadet-scanning
docker build -t riscv-gadget-scan .

# Get the kernel source.
git clone --depth 1 --branch v6.6 \
  https://github.com/torvalds/linux.git ~/linux-6.6
CACHE=mitigation_diffing/.embedding_cache
mkdir -p results .hf-cache "$CACHE"

# Run Smatch.
docker run --rm \
  -v ~/linux-6.6:/kernel \
  -v "$PWD/results:/opt/gadet-scanning/results" \
  riscv-gadget-scan \
  ./run-analysis.py linux-6.6 /kernel --tool smatch

# Run CodeQL. Creates a reusable database in
# results/linux-6.6/codeql-db/.
docker run --rm \
  -v ~/linux-6.6:/kernel \
  -v "$PWD/results:/opt/gadet-scanning/results" \
  riscv-gadget-scan \
  ./run-analysis.py linux-6.6 /kernel --tool codeql

# Run mitigation diffing.
docker run --rm \
  -v ~/linux-6.6:/kernel:ro \
  -v "$PWD/.hf-cache:/root/.cache" \
  -v "$PWD/$CACHE:/opt/gadet-scanning/$CACHE" \
  riscv-gadget-scan bash -lc "cd mitigation_diffing \
    && mitigation-diffing /kernel -m mitigations.csv"

# Compare tools (Smatch vs CodeQL).
```

```
docker run --rm \
  -v "$PWD/results:/opt/gadet-scanning/results" \
  riscv-gadget-scan \
  ./compare-tools.py linux-6.6
```

Expected results: the scanner emits kernel gadget reports; `mitigation-diffing` reports missing RISC-V mitigation counterparts for the listed cross-architecture mitigation sites. This should reproduce the findings in Section 4.1, where CodeQL and Smatch do not find the RISC-V specific gadgets that our mitigation diffing approach found.

(E5) Branchless uBPF dispatch benchmark [30 human-minutes + benchmark compute time].

```
cd branchless-jit-dispatch
make && ./dispatch
cd ubpf
cmake -S . -B build_baseline -DUBPF_ENABLE_TESTS=ON \
  -DCMAKE_BUILD_TYPE=Release
cmake --build build_baseline --target ubpf_test --parallel
cmake -S . -B build_mitigated -DUBPF_ENABLE_TESTS=ON \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_C_FLAGS="-DUBPF_JIT_DISPATCH"
cmake --build build_mitigated --target ubpf_test --parallel
./benchmark.sh build_baseline/bin/ubpf_test \
  build_mitigated/bin/ubpf_test \
  ../bpf-benchmark/bpf_progs 50
```

Expected results: `dispatch` runs the branchless direct-`jal` prototype. By inspecting the binary with `objdump -d` it can be confirmed that no indirect branches are used for the dispatch. The uBPF benchmark prints CSV rows `program,baseline_ms,mitigated_ms,overhead` which should reproduce the numbers in Section 5.2.

(E6, optional) SPEC CPU 2017 mitigation setup [2 human-hour + benchmark compute time]. Run it with a licensed SPEC CPU 2017 installation and a target sysroot.

```
cd specbuild
python3 -m venv .venv
. .venv/bin/activate
python3 -m pip install --upgrade pip pyyaml
docker build -t specbuild docker/
./fetch-sysroot.py <riscv-host> \
  ./sysroots/<riscv-host>-sysroot
# Place SPEC CPU 2017 at cpu2017/spec_cpu/
# or edit spec-build.yaml.
python3 build.py -l
python3 build.py -t <target> --size test \
  --benchmarks=505.mcf_r
scp packages/<target>-*.tar.gz <riscv-host>:~
# On <riscv-host>: unpack one package and run:
#   ./run.sh 505.mcf_r --csv spec.csv
```

Expected results: `specbuild` lists configured targets and fetches a sysroot from the target machine. The sysroot allows to cross compile the SPEC benchmarks for the target machine. The LLVM mitigation profiles additionally require the compiler bundle described in `specbuild/compilers/README.md`. The SPEC benchmarks should reproduce the overheads listed in Figure 3 of the paper.

A.5 Notes on Reusability

The artifact can be reused by testing all or a subset of the spectre variants on new RISC-V hardware. This requires adapting timers and the cache side channel to the new hardware, but should be much simpler than building a proof-of-concept from scratch. Our mitigation diffing can be used to map code sites from a kernel of one architecture to another. We believe that this is a useful tool not only for Spectre mitigations, but working with cross-architecture kernel code in general. The branchless JIT dispatch prototype could be extended into a more general mitigation by implementing it as part of a compiler pass.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: UncoreBleed: AEX-Free, High-Resolution, and Low-Noise Side-Channel Attacks on SGX Enclaved Execution

Decheng Chen, Zhi Zhang, Zhenkai Zhang, Xin Zhang, Yansong Gao, Yi Zou

A Artifact Appendix

A.1 Abstract

This artifact appendix describes the reproduction of the results from the paper *UncoreBleed: AEX-Free, High-Resolution, and Low-Noise Side-Channel Attacks on SGX Enclaved Execution*. This appendix includes the information needed to run the artifact and validate the claims made in the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

This artifact requires root privileges for PCIe configuration, CR0.CD modification, and kernel module loading. Evaluators should execute the artifact only on a dedicated experimental machine, since some steps may affect system stability or performance isolation. The artifact does not collect personal data and is intended solely for research and evaluation purposes on hardware owned or controlled by the evaluator.

A.2.2 How to access

The artifact is publicly available at the following stable URL: https://zenodo.org/records/17902416

A.2.3 Hardware dependencies

The following hardware is required:

- A 3rd, 4th, or 5th generation Xeon Scalable CPU with SGX support.
- Recommended processors:
  - Xeon Silver 4314 (Ice Lake-SP)
  - Xeon Bronze 3408U (Sapphire Rapids-SP)
  - Xeon Silver 4514Y (Emerald Rapids-SP)

A.2.4 Software dependencies

The following software is required:

- Linux kernel 5.4+ with SGX enabled (5.15 recommended)
- Intel SGX SDK 2.17+ installed and configured
- GCC 9.0 or newer
- CMake 3.16 or newer
- Root privileges for PCIe configuration, CR0.CD modification, and module loading

A.2.5 Benchmarks

This artifact does not rely on third-party benchmark suites. Instead, it includes custom workloads, demos, and experiment inputs developed for the evaluation of the claims in the paper. The artifact includes the following experimental components:

- A microbenchmark for comparing core and uncore PMCs during enclave and non-enclave execution.
- A PKT_MATCH analysis workload for studying programmable address filtering and 64-byte granularity.
- An address-bit flipping workload for reverse engineering physical-address-to-M2M mapping.
- A Libjpeg-based demo for recovering pictures from SGX enclaved execution.
- An RSA demo with TLBlur for demonstrating private-key extraction.

A.3 Set-up

A.3.1 Installation

Download the artifact from:

https://zenodo.org/records/17902416

Then extract the package and enter the root directory.

A.3.2 Basic Test

Run the following command in the root directory:

```
./detect_cpu.sh
```

If your CPU belongs to one of the supported generations, the script will print a message indicating compatibility. Otherwise, your platform may not support the artifact.

A.4 Evaluation workflow

A.4.1 Major Claims

- (C1): SGX enclaves are not entirely excluded from performance monitoring. While core PMCs are disabled, uncore PMCs remain active and expose enclave-correlated events.
- (C2): The uncore `PKT_MATCH` event supports programmable address filtering and achieves 64-byte granularity, enabling fine-grained monitoring of enclave memory accesses.
- (C3): The reverse-engineered `PKT_MATCH` event is consistently available across multiple generations of SGX-capable Xeon processors, despite being undocumented or inaccurately described in Intel manuals.
- (C4): As the first PMC-based side channel against SGX, UncoreBleed successfully recovers pictures during Libjpeg decompression, and extracts RSA private keys from a single decryption in production enclaves in the presence of TLBlur, the most recent software defense on off-the-shelf SGX platforms.

A.4.2 Experiments

(E1) Core vs. Uncore PMC Comparison [2 human-minutes + 10 compute-minutes]
This experiment validates that uncore PMCs remain active during SGX enclave execution, while core PMCs are largely suppressed. It supports claim (C1).

Preparation: Navigate to the `MicroBenchmark/` directory and read the `README.md` file. Ensure that the machine satisfies the SGX and privilege requirements described above.

Execution: Run the benchmark using `./run.sh` to sample both core and uncore events in enclave and non-enclave settings. After sampling completes, run `./summarize_perf.py` to summarize the collected measurements.

Results: The evaluator should compare the summarized event counts between enclave and non-enclave executions. The expected outcome is that core events such as `INSTRUCTIONS` and `BRANCHES` differ significantly between the two settings, while uncore events such as `TOR_INSERT`, `TAG_HIT`, `ACT_COUNT`, and `CAS_COUNT` remain within the same order of magnitude.

(E2) PKT_MATCH Event Analysis [1 human-minute + 10 compute-minutes]
This experiment reverse engineers the `PKT_MATCH` event and validates its filtering mechanism. It supports claim (C2).

Preparation: Navigate to the `EventAnalysis/` directory and read the `README.md` file. Confirm that the platform allows configuring and sampling the relevant uncore events.

Execution: Run `./Analysis` to generate high-frequency read/write operations, configure the `PKT_MATCH` filter, and collect event-count data. After the data is collected, run `./analyze_record.py` to analyze the resulting records.

Results: The evaluator should inspect the analyzed event-count output. The expected outcome is that `PKT_MATCH` reports DRAM traffic only when read/write operations occur near the physical address configured in the filter, with an effective granularity of 64 bytes.

(E3) Uncovering Address-to-M2M Mapping [5 human-minutes + 15 compute-minutes]
This experiment reverse engineers the physical-address-to-M2M mapping on the target platform. It supports claim (C3).

Preparation: Navigate to the `ReverseM2MMap/` directory and read the `README.md` file. `README.md` file.

Execution: Run `./ReverseMap.py`. This script flips selected address bits and monitors `PKT_MATCH` observations across M2M boxes in order to infer how memory accesses are routed.

Results: The evaluator should examine the output mapping generated by the script. The expected outcome is a recovered physical-address-to-M2M mapping consistent with the mapping pattern reported in Figure 6 of the paper.

(E4) UncoreBleed Attack on SGX-Based Libjpeg [15 human-minutes + 30 compute-minutes]
This experiment demonstrates picture recovery from enclaved Libjpeg execution using UncoreBleed. It supports claim (C4).

Preparation: Navigate to the `Demo/libjpeg/` directory and read the `README.md` file. Ensure that the SGX environment and all required dependencies are installed correctly.

Execution: Run `./run.sh`. The script automatically reads an example input image, captures the `PKT_MATCH` event stream, and processes the collected event counts to recover the memory-access trace.

Results: The evaluator should inspect the final recovered output produced by the script. The expected outcome is a recovered image qualitatively similar to the result shown in Figure 10 of the paper.

(E5) UncoreBleed Attack on RSA with TLBlur [30 human-minutes + 30 compute-minutes]
This experiment demonstrates RSA private-key extraction using UncoreBleed against a TLBlur-protected enclave. It supports claim (C4).

Preparation: Deploy TLBlur on the experimental platform. Follow the installation instructions provided by the TLBlur

open-source repository. Then navigate to the `Demo/RSA/` directory and read the `README.md` file.

Execution: Run `./run.sh`. The script captures `PKT_MATCH` events during RSA execution and then parses the recorded data to recover key-related information.

Results: The evaluator should inspect the script output and compare it with the paper. The expected outcome is a result qualitatively similar to Figure 11 of the paper, showing successful extraction of a RSA private key from a TLBlur-protected enclave.

A.5 Notes on Reusability

The artifact can also be reused to study uncore performance monitoring behavior across multiple Xeon generations, and to support further research on SGX side channels and uncore event reverse engineering.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Exploiting Hidden Resource Contention in Selective Speculation Defenses

Xiaoyu Cheng[1], Fei Tong[1,2,3] , Zhenyu Lei[1], Fang Jiang[1], Zhe Zhou[1], Guang Cheng[1], Trevor E. Carlson[4]

[1]School of Cyber Science and Engineering, Southeast University, China
[2]Purple Mountain Laboratories, China
[3]Jiangsu Province Engineering Research Center of Security for Ubiquitous Network, China
[4]National University of Singapore

A Artifact Appendix

A.1 Abstract

This artifact supports our study of reservation-station (RS) contention as a hidden leakage surface in optimized selective-speculation defenses. Specifically, it demonstrates that secrets can influence RS pressure through three mechanisms that are not fully covered by existing defense assumptions: operand-dependent μop expansion in `REP`-prefixed string instructions, predicated value selection without fetch redirects, and variable-latency arithmetic or memory operations that are left to proceed under optimized policies. These mechanisms can affect frontend fetch and I-cache state during transient execution, creating measurable side channels even when defenses delay known transmit operations or prediction-resolution events.

The package includes gem5-based PoCs that bypass STT and DOLMA, real-machine measurements validating the `REP MOVSB`/`REP STOSB` contention primitive on Intel Raptor Cove P-cores, and a bit-recovery PoC. It also provides the LLVM-based gadget-searching workflow, an optional libsodium timing-channel PoC, the enhanced STT implementation, and the SPEC CPU2017 workflow used for overhead evaluation.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact contains Spectre-style PoCs and microarchitectural timing experiments. It should be run only on private research machines where the evaluator is authorized to run low-level benchmarks. It does not include a privilege-escalation payload or intentionally read third-party data. Some real-machine experiments require root-only steps such as loading `msr` or nanoBench, disabling ASLR in one shell, and pinning processes to cores; these steps should not be performed on production systems.

A.2.2 How to access

The artifact is available at https://doi.org/10.5281/zenodo.20364720. We also provide the source code for the artifact at https://github.com/Arch-security/RSbypassSP.

A.2.3 Hardware dependencies

The gem5 simulator workflows in `gem5/stt/`, `gem5/dolma/`, and `gem5/gem5-sttre-enhanced/`, as well as the LLVM gadget-searching workflow in `gadget_detect/LLVM_FIX/`, do not require a specific CPU model. A Linux x86-64 host with Docker, at least 32 GB RAM, and tens of GB of free disk space is recommended for building gem5 and running simulations.

The real-machine validation in `Real-Machine_Validation/` requires an Intel CPU with Raptor Cove P-cores; our reference systems are an Intel Core i9-13900K and an Intel i7-14650HX running Ubuntu 22.04. The optional `gadget_detect/libsodium_poc/` timing PoC is hardware sensitive; we validated it on an Intel Xeon Gold 6248R and treat it as optional supporting evidence.

A.2.4 Software dependencies

The gem5 workflows need Linux on x86-64, Docker, `git`, and `make`; the Dockerfiles install the tree-specific dependencies, including the legacy Python/SCons environments and the RISC-V toolchain used by DOLMA. The real-machine workflows need Linux on x86-64, Bash, `make`, GCC, Python 3, `matplotlib`, `numpy`, `seaborn`, and `scipy`; nanoBench counter regeneration additionally requires root, `msr`, and the nanoBench kernel module. The LLVM gadget-searching workflow needs LLVM/Clang build dependencies, CMake, GNU Make, and source checkouts of the target libraries.

SPEC CPU2017 is required only for the enhanced STT overhead workflow and is not included in the artifact.

For convenience, the main artifact folders provide dependency helpers and the corresponding `requirements.txt` files record the Python package versions used in our tested environments.

A.2.5 Benchmarks

The attack PoCs are self-contained microbenchmarks. Gadget searching builds OpenSSL, Libgcrypt, libsodium, OpenBLAS, musl, zlib, and wolfSSL. The performance-overhead experiment uses evaluator-provided SPEC CPU2017.

A.3 Set-up

A.3.1 Installation

Clone the artifact repository and enter the artifact root:

```
git clone <artifact-url> RSbypassSP
cd RSbypassSP
export ARTIFACT_ROOT=$PWD
```

The component setup helpers are:

- `./gem5/install.sh` builds all three gem5 Docker images; pass `stt`, `dolma`, or `enhanced` to build only one.
- `./Real-Machine_Validation/install.sh` installs the normal real-machine dependencies on apt-based Ubuntu systems; nanoBench still requires the root-only setup in its README.
- `./gadget_detect/install.sh` installs dependencies for gadget detection and the optional libsodium PoC.
- For enhanced STT overhead, install SPEC CPU2017 separately and set `SPEC_DIR`.

A.3.2 Basic Test

The recommended smoke test is the STT `v1rep` run, which checks the Docker image, gem5 build, PoC build, and one complete simulator execution:

```
cd $ARTIFACT_ROOT
./gem5/install.sh stt
docker run --rm -it \
-v "$ARTIFACT_ROOT":/artifact
stt-ubuntu20-py2
```

Inside the container, run:

```
export ARTIFACT_ROOT=/artifact
cd $ARTIFACT_ROOT/gem5/stt
make x86-64
cd RS_Contention_Taxonomy-overlooked
make clean
make CC=musl-gcc all
cd $ARTIFACT_ROOT/gem5/stt
make v1rep
```

A successful run prints the Futuristic STT configuration and stable bit/timing lines for the `REP MOVSB`-based PoC.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Operand-dependent μop expansion creates pre-execution RS pressure that is not covered by existing execution-phase transmit taxonomies. Our gem5 `REP MOVSB`-based PoC shows that this pressure can bypass STT-Futuristic by affecting frontend/I-cache timing before STT's issue-time delay protection can apply. This is proven by (E1), described in **Section 5 Exploiting taxonomy-overlooked μop expansion under STT (Limitation I & II)** with results illustrated in **Figure 9** of the paper.

(C2): Variable-latency memory operations that are left to proceed under the older-μop-first allocation strategy can still create secret-dependent RS pressure when amplified by dependency chains. Our gem5 PoC shows that this leakage persists under DOLMA-Conservative (M+R) despite its older-μop-first allocation strategy. This is proven by (E2), described in **Section 5 Bypassing older-μop-first with taxonomy-recognized transmit μops under DOLMA (Limitation III)** with results illustrated in **Figure 10** of the paper.

(C3): Predicated value selection can inject secrets into operands without triggering fetch redirects, which is not covered by the delay-until-resolution mechanism. Our x86 `CMOV` and RISC-V Zicond PoCs show that such selectors leak under DOLMA-Conservative (M+R), while the branch-based selector is blocked. This is proven by (E2), described in **Section 5 Implementation-dependent effectiveness of delay-until-resolution** with results illustrated in **Figure 13** of the paper.

(C4): The `REP`-based RS-contention primitive is measurable on Intel Raptor Cove P-cores and supports bit recovery. This is proven by (E3), described in **Section 4.1** and **Section 5 Contention thresholds of `REP`-prefixed string instructions on Raptor Cove** with results reported in **Table 2** and **Figures 14–16** of the paper.

(C5): RS-contention-relevant gadget patterns occur in common libraries, and the optional libsodium PoC demonstrates a real-library timing example. This is proven by (E4) and (E6), described in **Section 4.3** with results reported in **Table 4** of the paper.

(C6): Enhanced STT closes the identified gaps by covering tainted stores, operand-dependent expansion, predicate sources, and conservative taint initialization for both memory- and register-resident secrets with moderate overhead. This is evaluated by (E5), described in **Section 6** with results illustrated in **Figure 17** of the paper.

A.4.2 Experiments

(E1): STT bypass through operand-dependent micro-op expansion *[0.1 human-hours + 0.2 computer-hours].*
How to: Reproduce the `REP MOVSB` STT bypass in gem5.
Preparation: Build the STT image with `gem5/install.sh stt`; inside the container, build `build/X86/gem5.opt` and `RS_Contention_Taxonomy-overlooked/`.
Execution: Run `make v1rep` from `gem5/stt/`.
Results: The output should show distinguishable bit-dependent timing under Futuristic STT, matching Figure 9.

(E2): DOLMA bypass through predicated and variable-latency transmitters *[0.2 human-hours + 0.3 computer-hours].*
How to: Reproduce (i) a variable-latency memory gadget that leaks despite DOLMA's older-μop-first allocation, and (ii) a selector comparison showing that a branch-based selector is blocked while x86 `CMOV` and RISC-V Zicond predicated selectors leak.
Preparation: Build the DOLMA image with `gem5/install.sh dolma`; inside the container, build the x86 and RISC-V gem5 targets and the PoCs under `RS_Contention_Taxonomy-recognized/`.
Execution: Run `make s1rs64`, `make s1rs64_branch`, `make s1rs64_cmov`, and `make s1rs_riscv`.
Results: The variable-time-memory PoC should leak under DOLMA-Conservative (M+R). The branch-based selector should be blocked by delay-until-resolution, while the x86 `CMOV` and RISC-V Zicond predicated selector PoCs should leak, matching Figures 10 and 13.

(E3): Contention thresholds of REP-prefixed string instructions on Raptor Cove *[0.3 human-hours + 0.5 computer-hours].*
How to: Validate the `REP MOVSB`/`REP STOSB` contention primitive and bit-recovery PoC on Raptor Cove.
Preparation: On a Raptor Cove system, run `Real-Machine_Validation/install.sh` and build `rcx_threshold_eval/` and `REPmsAttack/`.
Execution: Run `run_measuret.sh` and `analyze.py` for threshold selection, then run `analyze_chars.sh` with an evaluator-selected one-byte secret.
Results: The threshold sweep should show that larger RCX values for `REP MOVSB` and `REP STOSB` produce higher target-instruction latency after crossing a machine-dependent threshold. The attack script should recover the configured one-byte secret by majority vote. The generated plots should show distinguishable timing distributions for secret bits.

(E4): LLVM gadget search in common libraries *[0.3 human-hours + about 0.7 computer-hours].*
How to: Reproduce the static gadget-searching workflow for common x86 libraries.
Preparation: Run `gadget_detect/install.sh`; build LLVM/Clang `llvmorg-21.1.5` with `gadget_detect/LLVM_FIX/X86MIRanalyze_pass.patch`.
Execution: Compile the libraries listed in `LLVM_FIX/README.md` with `-O2 -mllvm -x86-mir-analyze -g`, then run `scripts/collect_gadget_table.sh`.
Results: The collected gadget-count table should match Table 4.

(E5): Enhanced STT SPEC2017 overhead *[0.6 human-hours + 50.3 computer-hours].*
How to: Reproduce the enhanced STT performance-overhead workflow.
Preparation: Build the enhanced image with `gem5/install.sh enhanced` or use `install_dependencies_ubuntu22.sh`; build `build/X86/gem5.opt`, install SPEC CPU2017, and set `SPEC_DIR`.
Execution: Use `run_spec17.sh` for a smoke run or `run_spec17suit.sh` for the paper-style configurations.
Results: Compute normalized IPC from `m5out/stats.txt`; the overhead trend should match Figure 17.

(E6): Optional libsodium real-library timing PoC *[0.1 human-hours + 0.1 computer-hours].*
How to: Run the optional real-library timing PoC for libsodium Ristretto255.
Preparation: Run `gadget_detect/install.sh`, build `gadget_detect/libsodium_poc/`, enter the ASLR-disabled shell, and resolve libsodium runtime addresses.
Execution: Rebuild if addresses changed, run `./mul 0` and `./mul 1`, and plot the logs.
Results: The two prepared microarchitectural states should produce visibly different aggregate timing distributions; exact cycles are machine dependent.

A.5 Notes on Reusability

The Dockerized gem5 PoCs are the most portable reproduction path. The LLVM pass can be reused on additional x86 libraries by compiling them with `-mllvm -x86-mir-analyze`. For real-machine experiments, rerun the threshold sweep whenever the CPU, kernel, BIOS setting, frequency policy, or core placement changes.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: What Adults Will (and Won't) Do to Prove Their Age: Empirical Evidence from a Deceptive Web Experiment

Yanzi Lin[*] Cheng Zhang[*] Madelyne Xiao[†] Lorrie Faith Cranor[*] Sarah Scheffler[*]

[*]Carnegie Mellon University [†]Princeton University

A Artifact Appendix

A.1 Abstract

This artifact package supports a web experiment that measures how adults respond to age verification systems when deciding whether to access age-restricted content. It contains the following components:

- Study platform: a React and Firebase web application that randomly assigned participants to one of seven age verification conditions, presented the corresponding interface, and logged their activity during the experiment.
- Demo video: a recording (`study_flow.mov`) showing the flow of the participants through the web experiment.
- Survey data and analysis code: anonymized CSV data from the experiment and post-task survey, along with R and Python scripts that generate all statistical results and figures reported in the paper.
- A complete appendix containing the demographics of the survey participants, complete statistical test results, screenshots from the study site, screenshots of the AgeGuardian interface for all seven conditions, recruitment ad designs and targeting parameters, the survey instrument, and the qualitative codebook.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Evaluators face no security risks when executing the artifact.

With respect to data privacy, the platform runs locally using placeholder Firebase credentials, does not perform real verification, and does not collect any personally identifiable information that the evaluators may have provided when interacting with the simulated age verification service.

The default Qualtrics survey links are live and are owned by the research group. Completing the survey will submit responses to the research group's Qualtrics account. Evaluators should review Qualtrics' privacy policy and **avoid entering real personal information**. For isolated testing, evaluators can instead substitute their own survey URLs, as described in the README.

A.2.2 How to access

The artifact is available at `https://doi.org/10.5281/zenodo.20343090`. It contains two sub-directories:

- `experiment-software/`
- `survey-data-and-analysis-code/`

Each directory has its own README with setup and run instructions.

A.2.3 Hardware dependencies

The artifact should run on any modern laptop or desktop computer. A smartphone with a camera is needed to evaluate the QR-code scanning feature of the artifact.

A.2.4 Software dependencies

The study platform requires Node.js 18 or later, npm, and a modern web browser. For local development, we recommend Firefox or Safari; additional browser-specific notes are provided in the README. Optional data-collection functionality requires the Firebase CLI, a Firebase project, and a Qualtrics account.

The data-analysis scripts require Python 3 and the Python packages listed in `requirements.txt`, which can be installed and run via `setup.sh`. The statistical analyses additionally require R; running `setup.R` installs the required R packages.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

To install and run the study platform, download the repository, navigate to the `experiment-software` subdirectory, and run `npm install && npm start`. The platform will then be available at `http://localhost:3000`. Evaluators who wish to test data-collection functionality should additionally follow the "Configure Firebase" instructions in the README.

To perform data analysis, navigate to the `survey-data-and-analysis-code` subdirectory and run `./setup.sh`. This script creates a Python virtual environment, installs the dependencies, and registers a Jupyter kernel. For the R analyses, run `Rscript setup.R` to install the required packages.

A.3.2 Basic Test

To test the study platform, first run `npm start` from the `experiment-software` subdirectory. Then open the following example URL in Firefox or Safari:

```
http://localhost:3000/?user_id=test&c=2&p=rf
```

A successful run displays the landing page. Clicking "Begin Study" should load the consent form in an iframe. After the consent form is submitted, the Important Notice screen should appear, followed by the government-ID condition specified by `c=2`. Completing the age-verification flow, or clicking "Exit Study", should redirect to the survey.

To test the Python analysis, after running `setup.sh`, open `generate_graphs.ipynb` and run all cells. No input parameters are required beyond the anonymized data files already included in the repository. A successful run produces nine PNG files in the `graphs/` sub-directory, and they should match Figure 3 through Figure 7 in the paper.

To test the R analysis, after running `setup.R`, knit or run all cells in `RQ1-RQ2-analysis.Rmd`. No input parameters are required. A successful run prints the chi-squared test result (with $\chi^2 = 619.82$, $p < .001$) and the pairwise proportions test results, matching the statistic reported in the Appendix (Section A.2.2) of the paper.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The platform implements the seven verification conditions listed in paper's Table 1: checkbox, three government-ID reassurance variants, government-ID with liveness check, AI age estimation, and email age estimation. These conditions can be selected using the URL parameter `c=1` through `c=7`.

(C2): The study platform implements the three non-completion paths described in §3.1 of the paper: closing the browser tab, reaching the inactivity timeout, or clicking the "Exit Study" button.

(C3): Participants' age verification completion behavior varies significantly by condition. The reassurance statements did not affect completion.

(C4): Checkbox and email age estimation are rated significantly more comfortable than other methods; Participants perceived government-ID-based methods and AI age estimation as riskier than checkbox or email verification; Checkbox was rated as significantly less effective than all other methods; GovID methods were rated significantly less convenient than all other methods.

(C5): Participants who reported higher comfort and lower perceived risk had significantly higher odds of completing age verification; perceived effectiveness was not significantly associated with completion.

A.4.2 Experiments

(E1): Verification-condition flows *[10 human-minutes]*: This experiment evaluates claim (C1) by checking that the study platform implements all seven verification conditions listed in Table 1 of the paper.
Preparation: Start the development server as described in §A.3.1, and open the platform in Firefox or Safari. For testing: open each URL in a new incognito or private window. Otherwise, changing the value of c will keep loading the first condition tested, because the platform saves URL parameters to the browser's session storage so a participant keeps their assigned condition if their page refreshes mid-study.
Execution: For each value $N \in \{1, \ldots, 7\}$, open:
`http://localhost:3000/?user_id=test&c=N&p=rf`
Results: Each value of `c` should load a verification condition that matches the corresponding row of Table 1.

(E2): Non-completion paths *[10 human-minutes]*: This experiment evaluates claim (C2) by checking that the study platform implements the three non-completion paths described in the paper.
Preparation: Start the development server as described in §A.3.1. To verify event logging, configure a Firebase project by following the README's "Configure Firebase" instructions; otherwise, the evaluator can still verify the user-visible behavior of each non-completion path locally.
Execution: Open any condition, for example:
`http://localhost:3000/?user_id=test&c=2&p=rf`
Then exercise the following three paths, each time with a fresh run: click "Exit Study", leave the tab idle until the inactivity countdown completes, and close the browser tab before completion.
Results: If "Exit Study" is clicked, the page should be

redirected to the exit survey. For tab closure or timeout, if configuration step 7 in README is performed, the email account entered in the consent form should receive an email containing a link to the survey. If Firebase is configured, the corresponding participant event documents should appear according to the data-collection schema described in the README.

(E3): Experiment data analysis *[5 human-minutes]*: This experiment evaluates claim (C3) by checking that the participants' age verification behavior matches the result reported in Table 3, Section A.2.2 in the Appendix.
Preparation: Ensure R and the required packages are installed by running `setup.R`. Confirm that `anonymized_experiment_data.csv` is present in the same directory as `RQ1-RQ2-analysis.Rmd`.
Execution: Open `RQ1-RQ2-analysis.Rmd` in RStudio and knit it (or run all chunks in order from the command line).
Results: Inspect the chi-squared test output and the pairwise proportion test output and confirm that they match with what is reported in Table 3 (Section A.2.2 in the Appendix).

(E4): Likert construct analysis *[10 human-minutes]*: This experiment evaluates claim (C4) by checking that participants' self-reported comfort, risk, effectiveness, and convenience ratings match the results reported in Tables 2, 4, 5, and 6 in the Appendix.
Preparation: Ensure R and the required packages are installed by running `setup.R`. Confirm that `anonymized_survey_data.csv` is present in the same directory as `RQ3-likert-analysis.Rmd`.
Execution: Open `RQ3-likert-analysis.Rmd` in RStudio and knit it (or run all chunks in order from the command line).
Results: Inspect the four Kruskal-Wallis test outputs (comfort, risk, effectiveness, convenience) and their pairwise Wilcoxon rank-sum test outputs, and confirm that they match what is reported in Tables 2, 4, 5, and 6 (Appendix A.2.1, A.2.3–A.2.5).

(E5): Completion regression *[5 human-minutes]*: This experiment evaluates claim (C5) by checking that the logistic regression of completion on comfort, risk, and effectiveness matches the result reported in Table 11 in the Appendix.
Preparation: Ensure R and the required packages are installed by running `setup.R`. Confirm that `anonymized_survey_data.csv` is present in the same directory as `completion_and_recruitment_regressions.Rmd`.
Execution: Open `completion_and_recruitment_regressions.Rmd` in RStudio and knit it (or run all chunks in order from the command line). The first chunk fits the logistic regression of completion on comfort, risk, effectiveness, and condition; the second chunk fits a separate regression on recruitment platform and gender (Appendix A.1) and is not part of this claim.
Results: Inspect the output of the first model ("LOGISTIC REGRESSION: COMPLETION COMFORT + RISK + EFFECTIVENESS + CONDITION") and confirm the results match Table 11 in the Appendix.

A.5 Notes on Reusability

The study platform is designed to be extensible. Researchers may adapt it to evaluate additional age-verification methods by adding new verification-condition components and routing parameters. The platform can also be adapted to other realistic study contexts.

The analysis code may also be reused for similar survey-based studies.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: XCFI: Comprehensive Control-Flow Integrity for Arm TrustZone-M

Yunju Gu
Pusan National University
ventus07@pusan.ac.kr

Jaeyeol Park
Pusan National University
woduf0628@pusan.ac.kr

Donghyun Kwon *
Pusan National University
kwondh@pusan.ac.kr

A Artifact Appendix

A.1 Abstract

This artifact provides the prototype of XCFI, a comprehensive control-flow integrity (CFI) mechanism for Arm TrustZone-M (TZ-M). Built on the Armv8.1-M PACBTI extension, XCFI uses a 32-bit control-flow identifier (CID) to enforce fine-grained CFI across the secure and non-secure states, covering indirect branches, returns, exceptions, and cross-state control-flow events. The artifact includes our modified LLVM toolchain, the Zephyr/TF-M software stack, the benchmarks, the ported ret2ns attack, and the scripts to build, flash, and run them. The experiments run on an Arm MPS3 FPGA board (AN555 image, Armv8.1-M Cortex-M85). The artifact is available on GitHub and archived on Zenodo.

A.2 Description & Requirements

XCFI is built with our modified LLVM toolchain and integrated with Zephyr and Trusted Firmware-M (TF-M); both secure and non-secure code is compiled with the modified toolchain and run on an Arm MPS3 FPGA board (AN555 image, Cortex-M85). At this board is not generally available, we provided remote access to a pre-configured one.

A.2.1 Security, privacy, and ethical concerns

XCFI is a defensive CFI mechanism and poses no risk to the evaluator's machine; no security mechanism is disabled. The artifact includes a port of the public `ret2ns` proof-of-concept attack, which runs only against the protected firmware on the board, not the evaluator's host.

A.2.2 How to access

The artifact is available at https://github.com/cslab-pnu/XCFI and https://doi.org/10.5281/zenodo.20195744.

*Corresponding Author.

A.2.3 Hardware dependencies

XCFI requires an Arm MPS3 FPGA board with the AN555 image, providing an Armv8.1-M Cortex-M85 core with PACBTI, TZ-M, and the MPU enabled; measurements use the DWT cycle counter.

A.2.4 Software dependencies

The host runs Ubuntu 22.04. The artifact builds with our modified LLVM 21 and integrates Zephyr v4.3 and TF-M v2.2. On a clean Ubuntu host, `install_llvm.sh` installs the modified LLVM toolchain and `zephyr_setup.sh` installs the Zephyr/TF-M environment and all remaining dependencies.

A.2.5 Benchmarks

BEEBS (20 longest-running benchmarks), the Zephyr `tfm_integration` examples (`psa_crypto`, `tfm_ipc`, `tfm_secure_partition`), and the ported ret2ns attack.

A.3 Set-up

A.3.1 Installation

On a clean Ubuntu host, set up from scratch:

```
git clone https://github.com/cslab-pnu/XCFI.git
cd XCFI
./install_llvm.sh
./zephyr_setup.sh
cd XCFI-zephyr
```

A.3.2 Basic Test

From `XCFI-zephyr/`, build and flash the `tfm_ipc` example, which exercises the full toolchain (modified LLVM, Zephyr, and TF-M across the secure/non-secure boundary):

```
./build_with_tfm.sh \
    samples/tfm_integration/tfm_ipc xcfi
./flash_with_tfm.sh
```

Reboot the board from the configuration console (`picocom -b 115200 /dev/ttyUSB0`, then type `reboot`). On the application console (`picocom -b 115200 /dev/ttyUSB1`) the boot chain (BL2 → TF-M secure → Zephyr non-secure) completes and the output ends with:

```
*** XCFI enabled ***
TF-M IPC on mps3
...
```

The `*** XCFI enabled ***` line and the completed boot chain confirm that all required components are functioning.

A.4 Evaluation workflow

All experiments use the same build/flash workflow. A build is selected with a mode: `base` (unprotected baseline) or `xcfi` (full XCFI); the experiments below use `xcfi`. `build.sh`/`flash.sh` build and flash a single Zephyr image (no TF-M), while `build_with_tfm.sh`/`flash_with_tfm.sh` build and flash the full TF-M image set (BL2 + secure + non-secure). The board exposes two serial ports: `/dev/ttyUSB0` is the configuration console (used to reboot the board after each flash) and `/dev/ttyUSB1` carries the application output. For each experiment, open a `picocom -b 115200` session on each port, run the build/flash commands, type `reboot` on `ttyUSB0`, and observe `ttyUSB1`.

A.4.1 Major Claims

(C1): XCFI provides fine-grained intra-state CFI (forward-edge, backward-edge, and exceptions); non-secure benchmarks run correctly with XCFI enabled, demonstrating that the instrumentation preserves legitimate execution. Shown by (E1), corresponding Sections 4.2 and 6.3 of the paper.

(C2): XCFI extends CFI to cross-state control flow between the secure and non-secure states; a full TF-M workload involving security state transitions and exceptions runs correctly with XCFI enabled. Shown by (E2), corresponding to Sections 4.3 and 6.6 of the paper.

(C3): XCFI detects and blocks real cross-state control-flow hijacking attacks (`ret2ns`). Shown by (E3), corresponding to Section 7.1 of the paper.

A.4.2 Experiments

(E1): [1 human-minute + 2–3 compute-minutes per benchmark]:
How to: build and run a non-secure BEEBS benchmark with XCFI enabled; it runs to completion, showing intra-state instrumentation preserves correct execution.
Preparation: from `XCFI-zephyr/`, with both serial consoles open.
Execution:

```
./build.sh benchmark/beebs/bubblesort xcfi
./flash.sh
```

then type `reboot` on `ttyUSB0`. To run a different BEEBS benchmark, change the project argument.
Results: `ttyUSB1` shows the boot banner, the `*** XCFI enabled ***` line, and the benchmark's cycle count.

(E2): [1 human-minute + 2–3 compute-minutes per benchmark]:
How to: build and run a full TF-M workload (BL2 + secure + non-secure) with XCFI enabled; it runs across security state transitions and exceptions, showing cross-state instrumentation preserves legitimate cross-state control flow.
Preparation: as in E1.
Execution:

```
./build_with_tfm.sh \
    samples/tfm_integration/tfm_ipc xcfi
./flash_with_tfm.sh
```

then type `reboot` on `ttyUSB0`. The `psa_crypto` and `tfm_secure_partition` examples are selected by changing the project argument.
Results: on `ttyUSB1` the boot chain (BL2 → TF-M secure → Zephyr non-secure) completes, `*** XCFI enabled ***` appears, and the PSA services return their results, and the workload reports its cycle count (full output in the repository README), confirming XCFI does not interfere with legitimate cross-domain control flow.

(E3): [1 human-minute + 2–3 compute-minutes]:
How to: run the ported `ret2ns` attack against the secure boundary; XCFI must detect and stop it.
Preparation: as in E1.
Execution:

```
./build_with_tfm.sh \
    samples/tfm_integration/ret2ns xcfi ATTACK2
./flash_with_tfm.sh
```

then type `reboot` on `ttyUSB0`. `ATTACK1` selects the backward-edge (BXNS) attack; `ATTACK2` selects the forward-edge (BLXNS) attack. See the README for details.
Results: on `ttyUSB1` ends with `[XCFI] Control-flow violation detected in Secure world!` showing the attack is detected and stopped before control is transferred. (In `base` mode the attack instead succeeds and `attacker controlled` is printed.)

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine

Ruiyi Zhang, Tristan Hornetz, Daniel Weber, Fabian Thomas, Michael Schwarz
CISPA Helmholtz Center for Information Security

A Artifact Appendix

A.1 Abstract

This artifact provides instructions to reproduce the findings in *StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine*. Our paper introduces StackWarp, a vulnerability stemming from a hardware bug in the AMD Zen CPUs' stack engine that allows for deterministic stack-pointer manipulation across SMT threads. We provide a fuzzer to locate the undocumented Model Specific Register (MSR) bit controlling the stack engine, a Performance Monitoring Counter (PMC) analysis suite to verify this bit's function, and a Proof-of-Concept demonstrating the vulnerability.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

To prevent weaponization and due to the extreme difficulty of providing host-privileged access to SEV-enabled server hardware, this evaluation focuses on the underlying hardware bug rather than a full TEE compromise.

Evaluating this artifact involves writing to undocumented MSR bits, which can potentially lead to system freezes or instability.

A.2.2 How to access

The artifact is available on Zenodo: https://zenodo.org/records/19108605

A.2.3 Hardware dependencies

In our paper, we evaluate the sychronization bug across multiple generations of AMD Zen microarchitectures. While we expect all AMD Zen 1-5 microarchitecturse to be affected, the specific microarchitectures verified in our study include:

- AMD Ryzen 5 2500U (Zen)
- AMD Ryzen 5 3550H (Zen+)
- AMD EPYC 7252 (Zen 2)
- AMD Ryzen 7 5700G (Zen 3)
- AMD Ryzen 9 6900HX (Zen 3+)
- AMD Ryzen 7 PRO 7840U (Zen 4)
- AMD EPYC 9124 (Zen 4)
- AMD Ryzen 9 9950X (Zen 5)

A.2.4 Software dependencies

We use Linux Ubuntu 22.04 and 24.04 in our experiments. We expect the following dependencies: `cmake`, `libasmjit-dev`, `libzstd-dev`, `linux-headers-amd64`, and `msr-tools`.

A.2.5 Benchmarks

None

A.3 Set-up

A.3.1 Installation

1. Clone the repository and install the dependencies:

```
sudo apt install libasmjit-dev libzstd-dev
 msr-tools cmake linux-headers-$(uname -r)
```

2. Ensure the MSR driver is loaded:

```
sudo modprobe msr
```

3. Enable the unprivileged access to performance counters:

```
echo 2 | sudo tee /sys/devices/cpu/rdpmc
```

A.3.2 Basic Test

Before running the full evaluation, execute this script to ensure the hardware interfaces are accessible:

```
chmod +x ./test.sh; ./test.sh
# Expected Result:
 [+] MSR access: SUCCESS
 [+] RDPMC userspace: SUCCESS
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): StackWarp allows for manipulation of the stack pointer on sibling thread. This is proven by experiment (E1) whose results are illustrated in Figure 1.

(C2): Flipping this bit results in measurable changes on stack instructions. This is proven by the experiments (E2) whose results are reported in table 2.

(C3): Our fuzzer finds this undocumented MSR bit that affects the architectural state of the sibling thread. This is proven by the experiment (E3) described in Section 3.3.

A.4.2 Experiments

(E1): *StackWarp PoC [5 minutes]*: A minimal PoC that shows stack pointer manipulation with the MSR bit by calling an unreachable function.
How to: Navigate to the folder `poc`.
Preparation: Compile the PoC code and find two sibling cores.

```
make
# The sibling cores share the L1 and L2 cache.
lscpu -e
```

Execution: Pin the victim to a specific core, and toggle the MSR bit on the sibling core at another terminal:

```
Core1>
taskset -c <Core1> ./unreach

# Manually interrupt upon success.
Core2>
while true; do sudo bash -c 'CUR=$(rdmsr
 0xc0011029); DISABLED=$(printf "%x" $((0x$CUR
 | 524288))); ENABLED=$(printf "%x"
 $((0x$ENABLED & ~524288))); wrmsr -p <Core2>
 0xc0011029 0x$DISABLED; sleep 1; wrmsr -p
 <Core2> 0xc0011029 0x$ENABLED'; done
```

Results: The output above confirms that the synchronization bug is present on this machine. The return address was redirected to other slots on the stack.

```
Core1>
% taskset -c <Core1> ./unreach
Unreachable!!!
```

(E2): *PMC Verification [10 human-minute + 5 compute-minute]*:
How to: Navigate to the folder `ucode_instr_num`.
Preparation: TODO.

```
# Offline the SMT first to avoid crash:
echo 0 | sudo tee
 /sys/devices/system/cpu/cpu<CORE2>/online

# Disable Op Cache to reduce noices.
sudo bash -c 'CUR=$(rdmsr 0xc0011021);
 DISOP=$(printf "%x" $((0x$CUR | 32))); wrmsr
 -p <CORE1> 0xc0011021 0x$DISOP'
```

Execution: Run the PMC analysis with the stack engine on and off.

```
taskset -c <CORE1> ./test

sudo bash -c 'CUR=$(rdmsr 0xc0011029);
 DISABLED=$(printf "%x" $((0x$CUR | 524288)));
 wrmsr -p <CORE1> 0xc0011029 0x$DISABLED'

taskset -c <CORE1> ./test
```

Results: The Delta will be printed on the terminal.

(E3): *MSR Fuzzing [1 human-hour + 1 compute-hour]:*
How to: Navigate to the folder `architectural_tests`.
Preparation: Specify one pre-defined msr list in `msr_lists.h` and build the tests.

```
# Specify the msr list
#define MSR_LIST msr_list_amd_zen4
# Build
cmake -B build && make -j$(nproc) -C build
```

Execution: Insert the MSR access kernel module before running test programs:

```
cd build
sudo insmod msr_flipper.ko
# choose one of the test programs, e.g.,
sudo ./single_instructions
```

Results: The tool generates instruction sequences. It toggles the MSR bits and compares the execution state of the tested sequence.
Warning: the stackwarp bit (`0xC0011029`, bit 19) is included in the list and may cause the system to freeze. Be advised that system hangs may persist even after this bit is commented out, as MSR modifications inherently introduce stability issues.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: M-Step: A Single-Stepping Framework for Side-Channel Analysis on TrustZone-M

Cristiano Rodrigues[1], Marton Bognar[2], Sandro Pinto[1], and Jo Van Bulck[2]

[1]*Centro ALGORITMI, Universidade do Minho,* [2]*DistriNet, KU Leuven*

A Artifact Appendix

A.1 Abstract

This artifact provides *M-Step*, the first open and extensible single-stepping attack framework specifically designed for Arm TrustZone-M. The package includes the source code, tooling, and documentation required to reproduce the paper's experiments and key results, specifically: (i) the M-Step microbenchmarks; (ii) the leakage analysis of various microarchitectural components; and (iii) the end-to-end attacks on Mbed TLS. We release the M-Step framework with the all the core mechanisms and plugins defined in the paper to enable future replication, engage the system security community, and allow future users to use M-Step as a Swiss army knife for side-channel analysis on Arm MCUs.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All the attacks are conducted on the target microcontroller, there is no security or privacy concerns for the host machine or the evaluators. The host machine is only used to obtain and analyze the results of experiments and running tools to help visualize the collected traces.

A.2.2 How to access

All artifacts are archived for long-term access on Zenodo at `https://doi.org/10.5281/zenodo.17910184`. Additionally, we also release them on GitHub as an open-source project at `https://github.com/M-Step-Framework`.

A.2.3 Hardware dependencies

Using M-Step and reproducing the experiments requires an STM32 Nucleo-144 development board (NUCLEO-L552ZE-Q), which includes the target MCU, an STM32L552ZE with an Arm Cortex-M33 core.

A.2.4 Software dependencies

The artifacts have been successfully tested on Ubuntu 22.04 LTS, Ubuntu 24.04 LTS, and NixOS (version 25.05). While the artifact is expected to be compatible with most modern Unix-based systems, we recommend using one of the tested distributions for guaranteed reproducibility.

Development Environment. To ensure a reproducible environment, we use Nix to automatically manage all required dependencies at their specific versions. The provided Nix development shell contains all the necessary tools to compile the source code, run the evaluation scripts, and use the third-party programs required by M-Step. For users on non-NixOS systems, Nix can be installed by following the instructions for the package manager at `https://nixos.org/download/`.

After installation, Nix Flakes and the new `nix` command must be enabled by adding the following line to the Nix configuration file (`~/.config/nix/nix.conf` or `/etc/nix/nix.conf`):

```
experimental-features = nix-command flakes
```

STM32 Programmer. The STM32 Programmer is required to flash binaries onto the target board. It can be downloaded from the official STMicroelectronics website (a free user account is required). After installation, ensure that the `STM32_Programmer_CLI` binary is available in your system's `PATH`. To grant the STM32 Programmer user-level permissions to access the board, create the file `/etc/udev/rules.d/99-stlink.rules` if it does not already exist, and add the following Udev rules. Without these rules, `sudo` will be required for every deployment.

```
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0483",
    ATTRS{idProduct}=="374e", MODE="0666",
    GROUP="plugdev"
SUBSYSTEMS=="usb", ATTRS{idVendor}=="0483",
    ATTRS{idProduct}=="374b", MODE="0666",
    GROUP="plugdev"
```

A.2.5 Benchmarks

None. The artifact is self-contained and does not require any external benchmarks or datasets.

A.3 Set-up

A.3.1 Installation

First, clone the artifact's source code from the Git repository and initialize its submodules:

```
git clone --recurse-submodules \
https://github.com/M-Step-Framework/m-step
cd m-step
```

Development Shell. All subsequent commands must be executed within the Nix development shell, which provides a reproducible environment with all necessary dependencies. To enter the shell, run the following command from the root of the repository. This command needs to be executed each time a new terminal session is started.

```
nix develop
```

A.3.2 Basic Test

To verify that the toolchain is functional and all dependencies are correctly installed, perform a basic end-to-end test of the M-Step framework. This test involves configuring, compiling, and deploying a simple application to the target board. Navigate to the `ns-world-bare` directory and execute the following steps:

1. Configure. Run the script to configure the Non-Secure and Secure world projects:

```
./0-config.sh
```

2. Compile. Compile the source code for both worlds:

```
./1-compile.sh
```

3. Deploy. Deploy the compiled binaries to the target board:

```
./2-deploy.sh
```

4. Monitor. Reset the board and capture its output:

```
./3-monitor.sh
```

Upon successful execution, you should observe the board booting, first into the Secure world (i.e., TF-M) and then transitioning to the Non-Secure world, which will print the message "*Hello NS World!!*". This confirms that the development environment is set up correctly.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): M-Step is a reliable single-stepping framework for Arm TrustZone-M that can adapt to various victim code patterns and instruction types, consistently stepping over functions with a negligible number of multi-steps. This claim is substantiated by experiment E1, which reproduces the microbenchmarks from §6.1, with results presented in Table 5.

(C2): M-Step is an extensible framework with modular, on-demand plugins (§5) that facilitate side-channel analysis across different Arm Cortex-M microarchitectural components. We demonstrate that M-Step can establish covert channels by exploiting various microarchitectural leakages, as proven by experiments E2–E5, which reproduce the results reported in §6.2, Figure 6.

(C3): M-Step's high temporal resolution enables two end-to-end proof-of-concept attacks that extract RSA private keys from Mbed TLS's key generation and import functions in a single trace. This claim is substantiated by E6, which reproduces the results reported in §6.3.

A.4.2 Experiments

(E1): *[M-Step Microbenchmarks] [5 human-minutes + 30 compute-minutes]: replicates the single-stepping reliability microbenchmarks from §6.1.*

Preparation: Navigate to the experiment directory:

```
cd evaluation/t1-mstp-metrics
```

Execution: Execute the evaluation script:

```
./1-run-test.sh
```

Results: The script will print the M-Step reliability metrics to the terminal, which can be compared against the results in Table 5 of the paper. The results are also saved to `./logs/0_raw_trace.txt` for later use.

(E2): *[DIV Instruction Covert Channel] [5 human-minutes + 2 compute-minutes]: replicates the data-dependent covert channel in the DIV instruction.*

Preparation: Navigate to the experiment directory:

```
cd evaluation/t2-covert-udiv
```

Execution: Execute the evaluation script:

```
./1-run-test.sh
```

Results: The plot, saved at `./logs/06-div.png`, demonstrates the covert channel and should resemble Figure 6a in §6.2.

(E3): *[Instruction Timing Covert Channel] [5 human-minutes + 2 compute-minutes]: replicates the covert channel based on varying instruction execution times.*

Preparation: Navigate to the experiment directory:

```
cd evaluation/t3-covert-inst
```

Execution: Execute the evaluation script:

```
./1-run-test.sh
```

Results: The plot at `./logs/06-inst_diff.png` demonstrates the covert channel and should resemble Figure 6b in §6.2.

(E4): *[ICache-based Covert Channel] [5 human-minutes + 2 compute-minutes]: replicates the instruction cache-based covert channel.*

Preparation: Navigate to the experiment directory:

```
cd evaluation/t4-covert-cache
```

Execution: Execute the evaluation script:

```
./1-run-test.sh
```

Results: The plot, saved at `./logs/06-cache.png`, demonstrates the covert channel and should resemble Figure 6c in §6.2.

(E5): *[Bus Contention Covert Channel] [5 human-minutes + 2 compute-minutes]: replicates the covert channel based on bus contention.*

Preparation: Navigate to the experiment directory:

```
cd evaluation/t5-covert-cont
```

Execution: Execute the evaluation script:

```
./1-run-test.sh
```

Results: The plot at `./logs/06-contention.png` demonstrates the covert channel and should resemble Figure 6d in §6.2.

(E6): *[End-to-End PoC Attacks] [5 human-minutes + 1 compute-hour]: replicates the two end-to-end attacks against Mbed TLS.*

Preparation: Navigate to the experiment directory:

```
cd evaluation/t6-pocs
```

Execution: Execute the evaluation script:

```
./1-run-test.sh
```

Results: The script will print the recovered RSA private keys for both proof-of-concept attacks to 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Memclave: Secure In-Memory Enclave for Untrusted Hosts

Amit Choudhari*
CISPA Helmholtz Center
for Information Security

Fabian van Rissenbeck*
TU Dortmund

Christian Rossow
CISPA Helmholtz Center
for Information Security

A Artifact Appendix

A.1 Abstract

The artifact contains the full source code of the Memclave framework, the ported Processing-In-Memory Benchmarks (PrIM) benchmark suite, and the plotting scripts for evaluation. Memclave includes (i) the Memclave *hypervisor* (`ci-switch` and a patched `QEMU`), (ii) a guest-side Linux kernel driver and In-memory Enclave (ime) client library that provide guest-side UPMEM-style abstractions, inside the guest VM, and (iii) the DRAM Processing Unit (DPU)-side runtime library (`ime`) contains Processing-in-Memory (PIM)-side subkernels, including the Memclave *loader*, First Stage Loader (FSL), key-exchange and messaging subkernels, and ported PrIM subkernels. The artifact additionally includes the unmodified PrIM benchmarks (native UPMEM baseline) and scripts required to reproduce the paper's plots and reported results.

A.2 Description & Requirements

The artifact is organized as follows.
`ci-switch`: A component of the Memclave hypervisor.
`qemu`: patched QEMU to interface with the CI-switch.
`driver`: Linux kernel module to interface with PIM.
`ime`: all DPU-side code (*loader*, FSL, key exchange and messaging subkernels, and the ported PrIM subkernels).
`ime-client-library`: guest-side library providing UPMEM-like abstractions and ported PrIM benchmarks.
`common`: shared headers used by Memclave components.
`mbedtls`: crypto library used both on-DPU and guest.
`plotting`: scripts used to generate figures for the paper.
`prim-benchmarks`: baseline PrIM suite for native UPMEM.
`scripts/hyp`: scripts for setting up the Memclave hypervisor
`README.md`: documentation and Reproduction guide.

Provided environments. Artifact contains (a) a VM image for execution and (b) two Docker images for development.

A.2.1 Security, privacy, and ethical concerns

The artifact is a confidential computation framework without any malicious components. We recommend evaluators execute the artifact on the provided test machine (Section A.2.3). It self-contained with benchmarking workloads.

A.2.2 How to access

We recommend evaluators use the Zenodo record (`https://doi.org/10.5281/zenodo.17986461`) to download the artifact. The source is mirrored on GitHub at `https://github.com/fabianvanrissenbeck/memclave`.

A.2.3 Hardware dependencies

For evaluation of Memclave, we need two linux systems (Sys_{Build} and Sys_{Run}). Sys_{Build} is a local linux based machine of the reviewer on which the docker image is deployed for building subkernels and ci-switch. Sys_{Build} does not have any specific configuration requirement as it is only used for building. Sys_{Run} is a remote machine with UPMEM PIM DIMMs (the paper evaluation runs on single PIM rank with 64 DPUs). To access the Sys_{Run}, SSH using the following credentials:

```
ssh -p 2293 reviewer@ios.inf.uni-osnabrueck.de
password: <Refer_HotCRP>
```

Tested configuration (from the paper, for reference): Sys_{Build}: intel Ultra 7 155H, Ubuntu 24.04.3 LTS (linux 6.11.11) Sys_{Run}: dual-socket Intel Xeon Silver 4216, Linux `5.15.0-142-generic`, UPMEM SDK `upmem-2025.1.0`, DPU frequency reported as 350 MHz per rank at runtime.

A.2.4 Software dependencies

Sys_{Run} has following requirements (already met). (1) x86 based Linux host with KVM support, (2) UPMEM SDK `upmem-2025.1.0`, (3) Toolchain for building, (4) Python 3.

Sys_{Build} has following requirements (pre-setup in the docker images). (1) Docker installed (to use the provided development images), (2) UPMEM SDK `upmem-2025.1.0` with a patched LLVM, (3) Toolchain for building the components (`gcc/g++`, `make`), (4) Python 3.

A.2.5 Benchmarks

The artifact evaluates Memclave using:

PrIM benchmark suite workloads. The PrIM suite contain 16 benchmarks, covering heterogeneous memory-access, compute, and synchronization patterns (native UPMEM baseline + Memclave-ported versions). The baseline suite is under `prim-benchmarks`, while Memclave ported guest-side and DPU-side components in `ime-client-library` and `ime`.
Microbenchmarks. Evaluate the subkernel load/unload, keyexchange, throughput for sealed data transfers and different transfer patterns (Table 3 and 4 from the main paper).

A.3 Set-up

We separate the artifact workflow into a development machine (Sys_{Build}), an execution machine (Sys_{Run}) and memclave environment (Sys_{Run_VM}). On Sys_{Build}, we build Memclave's host components and bundle them into `hyp/` using `./scripts/hyp/setup.sh`. We then copy `hyp/` and subkernels to Sys_{Run}. On Sys_{Run}, we launch the guest VM (Sys_{Run_VM}) and compile guest-side programs inside this VM.

A.3.1 Installation

Step 0: Obtain and the split of artifact. We download and unpack the artifact archive on both Sys_{Build} and Sys_{Run}. The directories relevant to Sys_{Build} are `ci-switch`, `hyp`, `ime`, `scripts/hyp`, `plotting`. And directories relevant to Sys_{Run} are `qemu`, `ime-client-library`, `prim-benchmarks`.

Step 1: Build hyp [on Sys_{Build}]. We use the docker image which includes build environment required to build PIM-side components and `ci-switch`. First, load docker images.

```
cd memclave-package
docker load -i memclave.tar
docker load -i memclave-qemu.tar
```

Build Hypervisor and copy to Sys_{Run}:

```
./scripts/hyp/setup.sh
scp hyp.tar.xz reviewer@<Sys_run>:~
```

Step 2: Boot and setup Sys_{Run_VM} [on Sys_{Run}]. On Sys_{Run}, extract transferred `hyp.tar.xz`, enter `hyp/` directory and run `boot.sh` specifying one from available Rank_IDs {0, 23, 26, 36}. Note: do not use {6, 7, 24, 25, 28, 31, 34, 35, 38, 39}:

```
cd hyp;
./boot.sh --exact-rank=<Rank_ID>
# login/password: memclave/memclave
```

Start `ci-switch` and launch the guest VM (Sys_{Run_VM}) using the patched QEMU. `setup-vm` loads `vpim` kernel module and setup reverse proxy. [Note]: Command runs in foreground and does not return. Keep the terminal open.

```
memclave@memclave-vm:~$ setup-vm
```

After loading, the driver creates device nodes `/dev/vpimN`.

Step 3: Setup reverse proxy to connect to Sys_{Run_VM} [on Sys_{Build}]. [Note]: This command stays in foreground and does not return. Keep the terminal open.

```
ssh -NL 26173:localhost:26173 -p 2293 \
    reviewer@ios.inf.uni-osnabrueck.de
```

Login to Sys_{Run_VM}.

```
ssh -p 26173 memclave@127.0.0.1
# user/pass: memclave/memclave
```

Step 4: Build subkernels [on Sys_{Build}] Build all Memclave subkernels and copy to Sys_{Run_VM}

```
cd ime; ./build.sh; cd ..
scp -r -P 26173 ime-client-library \
    memclave@127.0.0.1:~/
scp -P 26173 ime/subkernels.tar \
    memclave@127.0.0.1:~/ime-client-library/
```

Step 5: Build Memclave client apps [on Sys_{Run_VM}]. Build Memclave ported PrIM apps

```
cd ime-client-library
./build.sh
```

Step 6: Verify setup Confirm `boot.sh` successfully launches the Sys_{Run_VM}. The `setup-vm` successfully allocated ranks. Rank N, if allocated successfully, is mapped to /dev/vpimN. SSH to Sys_{Run_VM} is successful.

A.3.2 Basic Test

Run `bs` example on Sys_{Run_VM}.

```
cd build
./ime-bs-example
# Output: ...[OK] Outputs are equal
cd ..
```

A.4 Evaluation workflow

This section describes how to reproduce the key evaluation results of the paper (refer `reproduction.dox`).

A.4.1 Major Claims

(C1): Memclave is compatible with the full PrIM benchmark suite (Figure. 4). We port all 16 PrIM benchmarks and evaluate Memclave against CPU and native UPMEM (PIM-insecure) baselines. This is proven by experiment (E1), producing Figure. 4.

(C2): End-to-end overhead on representative workloads is modest. For MLP, Memclave converges toward the native UPMEM baseline as compute grows and incurs at most $\leq 1.5\times$ overhead at practical widths (Figure. 5); for BFS, overhead depends on graph/levels and is $\approx 1.1\times$ on some graphs (Figure. 6). Experiment (E2) proves this and generates Figure. 5 and Figure. 6.

(C3): Microbenchmarks reproduce the reported transfer costs and patterns (Tables 3–4). We reproduce (i) sealed-transfer and transfer-pattern throughput and (ii) profile subkernel loading and key exchange. Experiments (E3) and (E4) prove this and generate Tables 3–4.

A.4.2 Experiments

We provide a standard template (Run_Baseline, Run_Memclave, Plot) for running each experiment.

Run_Baseline: Run <Expt> **[on Sys_{Run}]**

```
cd memclave-package/prim-benchmarks
./run.sh <Expt: prim, bfsmlp, mram, subk, all>
# Output: output.tar
```

Run_Memclave: Run <Expt> **[on Sys_{Run_VM}]**

```
cd ime-client-library
./run.sh <Expt: prim, bfsmlp, micro, subk, all>
# Output: output.tar
```

Plot: Plots figures **[on Sys_{Build}]**. [Note]: Plotting dependencies are satisfied in `plotting/requirements.txt`

```
scp -P 2293 reviewer@ios.inf.uni-osnabrueck.de:\
~/output.tar output_baseline.tar
scp -P 26173 memclave@localhost:~/output.tar \
  output_memclave.tar
cd plotting; ./plot_all.sh \
  ../output_memclave.tar ../output_baseline.tar
```

(E1): PrIM suite compatibility (Reproduce Figure. 4) [10 human-mins + 10 compute-min].
Goal: Run all 16 PrIM benchmarks in native UPMEM and in Memclave; join results; generate Figure. 4 plot.
Step E1.1 : Run the baseline PrIM. Follow `Run_Baseline` with <expt> as `prim`
Step E1.2: Run the Memclave PrIM. Follow `Run_Memclave` with <expt> as `prim`
Step E1.3: Generate plots. Follow `Plot`.
Result: Figure. 4-style breakdown across all 16 benchmarks.

(E2): MLP and BFS overhead (Reproduce Figure. 5,6) [10 human-min + 10 compute-min]
Goal: run MLP and BFS on Memclave and native UPMEM and plot the comparison.
Step E2.1: Run MLP and BFS on UPMEM baseline. Follow `Run_Baseline` with <expt> as `bfsmlp`
Step E2.2: Run MLP and BFS on Memclave. Follow `Run_Memclave` with <expt> as `bfsmlp`
Step E2.3: Generate plots. Follow `Plot`.
Result: Generates Figure. 5, Memclave approaches PIM-insecure for larger N, with $\leq 1.5\times$ overhead at practical widths. Generates Figure. 6, comparison across graphs (LiveJournal1, loc-gowalla, roadNet-PA).

(E3): Sealed transfer and transfer-pattern throughput to/from PIM (Reproduce Table 3, 4) [10 human-mins + 30 compute-min].
Goal: build the crypto benchmark and the subkernel, then run the benchmark to emit CSV.
Step E3.1: Run baseline. Follow `Run_Baseline` with <expt> as `mram`
Step E3.2: Run Memclave. Follow `Run_Memclave` with <expt> as `micro`
Step E3.3: Generate plots. Follow `Plot`.
Result: CSV outputs corresponding to Table 3 and 4.

(E4): subkernel loading and key exchange (Reproduce Table 3) [10 human-mins + 10 compute-min].
Goal: Measure subkernel loading/unloading and key exchange timings.
Step E4.1: Run baseline. Follow `Run_Baseline` with <expt> as `subk`.
Step E4.2 [on $\text{Sys}_{\text{Run_VM}}$ *]:* Restart Sys_{Run_VM} for subkernel profiling.

```
sudo poweroff
```

Step E4.3 [on Sys_{Run} *]:* Rerun Installation Step 2 with boot-stats.sh.

```
cd hyp;
./boot-stats.sh --exact-rank=<Rank_ID>
# continue steps 2 from installation
```

Step E4.4 [on $\text{Sys}_{\text{Build}}$ *]:* Rerun Installation Step 3
Step E4.5: Run Memclave. Follow `Run_Memclave` with <expt> as `subk`
Step E4.6: Generate plots. Follow `Plot`.
Result: CSV outputs corresponding to 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Rarus: A Succinct and Efficient Range Proof for Polynomial-based Vector Commitment

Xinyang Yang, Wenjie Qu, Yanpei Guo, Jiaheng Zhang

A Artifact Appendix

A.1 Abstract

The artifact provides the complete C++ implementation of Rarus, a zero-knowledge polynomial range proof protocol, along with baseline comparisons (Bulletproofs and Plookup). It leverages the high-performance `mcl` library for underlying MSM and pairing functions. The artifact includes all necessary cryptographic gadgets, such as non-recursive Bivariate NTT, polynomial operations, and KZG commitments. It is designed to reproduce the core benchmarking claims made in the paper regarding Prover time, Verifier time, and Proof size.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

The stable and permanently archived version of this artifact is available on Zenodo at https://doi.org/10.5281/zenodo.20322978. The active development repository, which can evolve during the evaluation discussion phase, is hosted on GitHub at https://github.com/Montayang/rangeproof.

A.2.3 Hardware dependencies

None. The artifact runs entirely on standard CPU-based environments. It does not require GPUs, Trusted Execution Environments, or any custom hardware.

A.2.4 Software dependencies

The system is tested on standard vanilla Linux distributions (e.g., Ubuntu 22.04 LTS or 24.04 LTS). It requires standard C++ build tools and cryptographic libraries, which can be installed via the following commands:

```
sudo apt-get update
sudo apt-get install -y libgmp-dev libssl-dev
sudo apt-get install -y cmake make g++ git python3
```

The protocol strictly depends on the `mcl` library (V3+), which will be compiled from source during the set-up phase.

A.2.5 Benchmarks

The experimental evaluation relies on the baseline implementations of Bulletproofs and Plookup included in this artifact (`bulletproof.cpp`, `plookup.cpp`). No external datasets or pre-trained models are required. The benchmark executable automatically generates random polynomial inputs based on the parameters (b,k,ℓ) specified in Section 6 of our paper to measure the protocol's efficiency.

A.3 Set-up

A.3.1 Installation

1. Clone the Rarus repository and enter the directory.
2. Clone the `mcl` library directly into the project root:
   `git clone https://github.com/herumi/mcl.git`
3. Compile the `mcl` core library:
   `cd mcl && make clean && make -j4 && cd ..`
4. Build the main project:
   ```
   mkdir build && cd build
   cmake .. && make -j4 && cd ..
   ```

A.3.2 Basic Test

Navigate to the `build` directory and execute the main binary: `./main`. The expected successful output will display a sequence of completion logs (e.g., `Setup Completed.`, `Step1 Completed.`), concluding with the Prover Time, Verifier Time, and Proof Size without any linking or segmentation faults.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Rarus achieves specific performance metrics (Prover Time, Verifier Time, and Proof Size) as reported in Section 6 of the paper. This is proven by the core experiment (E1).

A.4.2 Experiments

(E1): Automated Benchmarks [5 human-minutes + 30 compute-minutes + 500MB disk]: Evaluate the core protocol and baselines to reproduce the paper's results.
How to: Use the provided Python master script to automatically inject parameters, recompile, and execute the benchmarks.
Preparation: Complete all steps in the Set-up and Installation section to initialize the `build` directory and compile the `mcl` core library.
Execution: From the project root directory, run the master script: `python3 run_experiments.py`. The script handles incremental compilation and execution automatically.
Results: The terminal will output formatted ASCII tables that directly mirror the execution time (Prover/Verifier Time in seconds) and proof size (in KB) results presented in Table 8, Table 9, and the data points in Figures 3 and 4 of our paper.
Important Note on Evaluation Variance: As discussed during the AE process, evaluators should expect variations in absolute execution times. Because the underlying cryptographic operations (e.g., MSMs and NTTs) are heavily CPU-bound, absolute wall-clock times will naturally differ across various hardware architectures (e.g., differences in microarchitecture or clock speed). Additionally, standard Linux environments may exhibit run-to-run variance (typically 5%–15%) due to OS scheduling and thermal dynamics. However, the core claims of our paper—the *relative asymptotic scaling ratios* and the mathematically deterministic *Proof Sizes* (e.g., 1.53125 KB)—will remain strictly consistent across all environments.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Prezta: Provable Remote Execution of Zero-Trust Authorization using SNARKs

Zhongjing Wei, Osaid Muhammad Ameer, Nikita Borisov, Yupeng Zhang
University of Illinois Urbana-Champaign

A Artifact Appendix

A.1 Abstract

This artifact contains the source code and benchmarks for Prezta, a SNARK-based authorization architecture for Operational Technology (OT) systems. The artifact consists of two main components: (1) the Prezta prototype implementation using RISC Zero zkVM v3.0.3, including proof generation and verification; (2) an XACML-to-Rust compiler that translates XACML 3.0 access control policies into Rust code compatible with zkVM execution. The artifact allows reproducing all main experimental results reported in Section 5 of the paper, including end-to-end performance (Figures 1–2), ablation study (Table 2), and user cycles breakdown (Table 3).

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. The artifact is a research prototype that does not perform any destructive or privacy-violating operations.

A.2.2 How to access

The artifact is permanently archived on Zenodo (concept DOI, always resolving to the latest version):

https://doi.org/10.5281/zenodo.20303461

The repository is also available on GitHub at https://github.com/walotta/ZK_Zero_Trust. The main experiments (E1, E2) use the `master` branch. The ablation study (E3, E4) requires the `AblationStudy` branch, which is included in the repository.

A.2.3 Hardware dependencies

An x86_64 Linux machine with at least 32 GB RAM is required to reproduce the experiments. The paper's results were obtained on an AWS c7i.4xlarge instance (16 vCPUs, Intel Xeon Scalable, 32 GB RAM). No GPU is required; CUDA support is optional and can accelerate proof generation.

A.2.4 Software dependencies

- Ubuntu 22.04+ (or equivalent Linux distribution)
- System packages: `build-essential`, `pkg-config`, `libssl-dev`, `git`, `curl`
- Rust stable toolchain (via `rustup`)
- RISC Zero zkVM v3.0.3 (installed via `rzup`)
- Python 3.10+ with `pip` (packages: `lxml`, `Jinja2`, `cryptography`, `tqdm`, `maturin`)
- Nix (optional, provides reproducible environment via `flake.nix`)

A.2.5 Benchmarks

The XACML 3.0 conformance test suite is included in the repository under `tools/xacml-to-rust/policy_test_set/`. Running the compiler (E0) generates 368 policy code files with corresponding request/response pairs into `jwt_zkvm_testing/` (with RSA) and `zkvm_testing_without_rsa/` (without RSA). Pre-signed JWT tokens are stored in `test_jwts/` and automatically included in the generated datasets.

A.3 Set-up

A.3.1 Installation

```
# System dependencies (Ubuntu)
sudo apt update && sudo apt install -y \
  build-essential pkg-config libssl-dev \
  git curl python3-venv

# Install Rust toolchain
curl --proto '=https' --tlsv1.2 -sSf \
  https://sh.rustup.rs | sh -s -- -y
source "$HOME/.cargo/env"

# Install RISC Zero toolchain (pin to v3.0.3)
curl -L https://risczero.com/install | bash
source "$HOME/.bashrc"  # or restart shell
rzup install cargo-risczero 3.0.3
```

```
rzup install r0vm 3.0.3

# Clone main repository
git clone \
  https://github.com/walotta/ZK_Zero_Trust.git
cd ZK_Zero_Trust
mkdir -p logs

# Clone the XACML-to-Rust compiler
git clone \
  https://github.com/osaidameer/xacml-to-rust.git \
  tools/xacml-to-rust

# Install compiler dependencies
python3 -m venv ~/.venv
source ~/.venv/bin/activate
pip install lxml Jinja2 cryptography tqdm maturin

# Build regex compiler extension
cd tools/xacml-to-rust/compile_regex
PYO3_USE_ABI3_FORWARD_COMPATIBILITY=1 \
  maturin develop --features python
cd ~/ZK_Zero_Trust
```

A.3.2 Basic Test

After running E0 (see Section A.4), run the batch script on a single policy to verify the environment is correctly configured. First, temporarily limit execution to one test case:

```
sed -i '36 a testcases_names = ["IIA001"]' \
  batch_exec.py
python3 batch_exec.py
# Revert the modification
git checkout -- batch_exec.py
```

Expected: the log in `logs/` should contain `Total cycles: 262144`, `Gen time elapsed: ~28s`, and `Verify time elapsed: ~15ms`. The first run includes full compilation and may take 5–10 minutes. Completion is indicated by the message `All policies processed.` on stdout.

Note: We recommend running all long experiments inside `tmux` or `screen` to prevent SSH disconnections from interrupting execution.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Prezta's XACML compiler supports 83% of the XACML 3.0 conformance suite (322 out of 389 testable policies). This is proven by experiment (E1) and reported in Section 5.1.

(C2): Prezta's proof generation completes in ∼14–28 seconds per policy on a c7i.4xlarge instance, and verification takes 14–16 ms. This is proven by experiment (E1) and reported in Section 5.1.

(C3): Without RSA verification, 88.6% of policies achieve proof generation in ∼7 seconds. This is proven by experiment (E2) and reported in Figure 2.

(C4): Prezta's optimizations (RSA precompile + regex DFA pre-compilation) reduce prover time by 47.7× compared to a naive approach. This is proven by experiment (E3) and reported in Table 2.

(C5): The user cycles breakdown shows RSA contributes 45% and regex 47% of user cycles. This is proven by experiment (E4) and reported in Table 3.

A.4.2 Experiments

(E0): XACML-to-Rust Compilation [2 human-min + 2 compute-min]: Run the XACML-to-Rust compiler to generate Rust policy code from XACML sources. This step produces the datasets used by experiments E1 and E2.
Preparation: Ensure the Python virtual environment is activated (`source ~/.venv/bin/activate`).
Execution: `cd ~/ZK_Zero_Trust`
`bash run_compiler.sh true`
`bash run_compiler.sh false`
Results: Expected: ∼363 policies compiled successfully out of 389 attempted per configuration. The 26 failures are due to unsupported XACML higher-order functions (not used in practice). The generated datasets are placed directly in the directories that `batch_exec.py` reads from, so experiments E1 and E2 can be run immediately after this step.

(E1): End-to-End Performance [5 human-min + 2.5 compute-hours]: Run all conformance test policies with full Prezta (RSA + regex optimizations).
Preparation: Ensure E0 has been run with `bash run_compiler.sh true`. Ensure you are on the `master` branch.
Execution: `python3 batch_exec.py`
Completion is indicated by `All policies processed.` on stdout.
Results: The log is written to `logs/no_rsa_batch_<timestamp>.log`. The summary line reports total test cases and failures. Expected: 368 cases processed, ∼46 failures. The conformance suite contains 389 testable policies; 373 compile to Rust successfully; 368 have complete test inputs (request/response/JWT); 322 produce correct authorization decisions, yielding a coverage rate of 322/389 ≈ 83%. Note: 8 of these 322 involve malformed requests that fail deserialization; since the subject cannot furnish the required attributes, the correct decision is Deny (no proof needed), which aligns with the XACML conformance expectation. Extract metrics via:
`cp logs/<logfile> data/end2end/end2end.log`
`python3 data/end2end/extract.py \`

```
  -o data/end2end/metrics.csv
```
The resulting `metrics.csv` contains per-policy prover time (∼28s or ∼14s), verification time (14–16 ms), and proof size (238–250 KB), corresponding to Figures 1 and the text in Section 5.1.

(E2): End-to-End without RSA [5 human-min + 50 compute-min]: Run all policies with RSA verification removed to reproduce Figure 2.

Preparation: Ensure E0 has been run with `bash run_compiler.sh false`. On the `master` branch, change the dataset path in `batch_exec.py`:

```
sed -i 's/jwt_zkvm_testing/\
  zkvm_testing_without_rsa/' batch_exec.py
```

Execution: `rm -rf target/`

```
python3 batch_exec.py
```

Results: Expected: ∼89% of policies complete in <10s (∼7.3s), with total cycles of 65,536 (2^{16}). This matches the paper's claim that 88.6% drop to 7 seconds. After recording results, revert the modification:

```
git checkout -- batch_exec.py
```

(E3): Ablation Study [15 human-min + 35 compute-min]: Reproduce Table 2 by running policy IIC057 under three optimization configurations.

Preparation: Switch to the `AblationStudy` branch:

```
git checkout AblationStudy
```

The file `methods/guest/src/main.rs` contains three configuration constants at lines 16–18. The batch script is pre-configured to run only IIC057. The test input files are:

```
REQ=tools/xacml-to-rust/jwt_zkvm_testing/\
  requests/Policy_IIC057.json
RESP=tools/xacml-to-rust/jwt_zkvm_testing/\
  responses/Policy_IIC057.json
JWT=tools/xacml-to-rust/jwt_zkvm_testing/\
  jwts/Policy_IIC057.json
```

Execution: Run three configurations sequentially:

(a) "None" (naive, no optimizations): In `methods/guest/Cargo.toml`, comment out the `rsa` and `sha2` lines under `[patch.crates-io]`. In `methods/guest/src/main.rs`, set:

```
const USE_STD_REGEX: bool = true;
const BYPASS_REGEX: bool = false;
const BYPASS_RSA: bool = false;
```

Then run:

```
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Expected: ∼10.5M user cycles, ∼1300s (∼22 min).

(b) "+RSA" (RSA precompile only): Uncomment the `rsa` and `sha2` patches in `Cargo.toml`. Keep `USE_STD_REGEX=true`, `BYPASS_RSA=false`.

```
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Expected: ∼805K user cycles, ∼176s.

(c) "+Regex" (full Prezta): Set `USE_STD_REGEX=false`, `BYPASS_RSA=false`.

```
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Expected: ∼153K user cycles, ∼28s.

Results: Compare user cycles, prover time, verifier time, and proof size against Table 2. User cycles should match within 1%; prover times may vary by ∼10% due to machine load.

(E4): User Cycles Breakdown [5 human-min + 10 compute-min]: Reproduce Table 3 by isolating each module's contribution.

Preparation: On the `AblationStudy` branch, ensure `[patch.crates-io]` patches are active (uncommented).

Execution: Run IIC057 with both RSA and regex bypassed:

```
# In methods/guest/src/main.rs, set:
# const USE_STD_REGEX: bool = false;
# const BYPASS_REGEX: bool = true;
# const BYPASS_RSA: bool = true;
rm -rf target/
cargo run --release -- $REQ $RESP $JWT
```

Results: Expected user cycles ∼13,507 ("Others"). Combined with E2's IIC057 result (∼84,855 = Others + Regex) and E3c (∼153,513 = full), derive:

- Others = 13,507
- Regex = 84,855 − 13,507 = 71,348
- RSA = 153,513 − 84,855 = 68,658

These match Table 3 (paper: 13,508 / 71,429 / 68,598).

A.5 Notes on Reusability

The XACML-to-Rust compiler (`tools/xacml-to-rust/`) can be used to compile custom XACML policies for zkVM execution. New policies can be tested by placing the generated Rust code in `methods/guest/src/main.rs` and running the host binary. The system can also be adapted to other zkVMs (e.g., SP1, Zisk) by replacing the RISC Zero-specific APIs with equivalent calls.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Efficiently Provable Approximations for Non-Polynomial Functions

Sriram Sridhar[1] Shravan Srinivasan[2] Dimitrios Papadopoulos[3] Charalampos Papamanthou[2,4]

[1]University of California, Berkeley [2]Lagrange Labs
[3]Hong Kong University of Science and Technology [4]Yale University

A Artifact Appendix

A.1 Abstract

Zero-Knowledge Proofs (ZKPs) are now widely used to verify the correctness of various types of computations. However, despite phenomenal advancements, current ZKPs are inefficient for applications that need accurate evaluation of non-polynomial functions over floating-point numbers, such as machine learning, decentralized finance, scientific computing, and geolocation. Current state-of-the-art approaches typically emulate floating-point numbers using fixed-point representations (via quantization), and handle *non-polynomial* functions using lookup tables, piece-wise or low-degree polynomial approximations, which lead to sub-optimal performance and/or loss in accuracy or generality, limiting their potential for adoption in practice.

In this work, we present a general framework for approximating a large class of non-polynomial functions using Gauss-Legendre quadrature, which supports efficient ZKPs of correct computation. We show that our approach decreases the *adversarial error* (the maximum number of lower-order bits a cheating prover can manipulate undetected) up to the limits imposed by quantization, without increasing the multiplicative circuit depth beyond a small constant (≤ 4). This is a strong deviation from prior approximation techniques, where decreasing the adversarial error leads to increased multiplicative depth – the main factor determining the adversarial error growth of an approximation. We implement and evaluate our approach in Noir/Barretenberg, and we obtain absolute adversarial errors $2-256\times$ lower than comparable baselines for most non-polynomial functions with low prover overhead. We also demonstrate an efficient prover and low adversarial errors for high-accuracy applications in DeFi and astronomy that require non-polynomial functions, again obtaining adversarial errors $4-64\times$ lower than the baseline approximations.

This submission is the artifact for the above paper. It contains source code corresponding to the experiments in Tables 2-5 of the paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

No ethical concerns. The artifact does not collect personal data, access network resources, or disable any security mechanisms. It performs only local computation.

A.2.2 How to access

The artifact is available at https://zenodo.org/records/20246456.

A.2.3 Hardware dependencies

None. The artifact was tested on an Apple M3 Pro (12 cores: 6 performance + 6 efficiency) with 36 GB RAM, but should run on any machine with sufficient memory. No specialized hardware (GPU, FPGA, etc.) is required.

A.2.4 Software dependencies

- Python 3 (tested with 3.13)
- Python packages: `sympy==1.14.0`, `toml==0.10.2` (listed in `requirements.txt`)
- Noir (tested with `nargo` v1.0.0-beta.21), installed via `noirup`
- Barretenberg backend (tested with `bb` v5.0.0), installed via `bbup`

A.2.5 Benchmarks

None. All inputs are generated programmatically by the witness generation scripts.

A.3 Set-up

A.3.1 Installation

1. Clone the repository:

```
# Download and extract from Zenodo:
# https://zenodo.org/records/20246456
cd non-linear-artifact
```

2. Install Python dependencies:

```
pip3 install -r requirements.txt
```

3. Install Noir and Barretenberg:

```
noirup
bbup
```

A.3.2 Basic Test

Run a single experiment to verify the setup:

```
python3 scripts/gl_gen.py sigmoid 100 8 16
python3 scripts/run_prover.py gl_quad
```

This generates a witness for an order-8 Gauss-Legendre quadrature approximation of the sigmoid function at 16-bit fixed-point scale, then proves and verifies the computation using Noir/Barretenberg. A successful run prints proving key time, prover time, verifier time, and proof size.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Our microbenchmarks for the two approaches in our work: Padé and Gauss-Legendre quadrature approximations, vs existing approaches (polynomial, piecewise-linear, lookup table approximations) for a large class of non-polynomial functions. This is present in Table 2 of the paper.
(C2): For high-accuracy applications in DeFi and astronomy, we report prover times in Table 3.
(C3): In Tables 4,5 we compare circuit sizes of our fixed-point approaches vs floating-point based proof systems.

A.4.2 Experiments

The experiments correspond to the claims made above directly: **C1** to **E1**, **C2** to **E2** and **C3** to **E3**.
We estimate human time of about 15 minutes for setup and running the experiments, while the experiments themselves should take less than 30 minutes of compute (wall-clock) time on the hardware in Sec. A.2.3 (**E1** takes the longest at around 15-20 minutes, while **E2,E3** take less than 5 minutes each).

(E1): Run all microbenchmarks for Table 2 using `python3 scripts/run_all.py`. The results csv is stored in the `./experiments` folder. Note that some experiments marked as failed are by design and should match the empty entries in the paper; (these are either due to them taking unreasonably long or having large error, in which case we do not run them as mentioned in the paper).
(E2): Run the Orbital mechanics experiments using

```
python3 scripts/gl_gen.py kepler 1 30 120
python3 scripts/run_prover.py gl_quad
python3 scripts/poly_gen.py kepler 1 30 120
python3 scripts/run_prover.py poly
python3 scripts/pade_gen.py kepler 1 16 120
python3 scripts/run_prover.py pade
```

For the Bancor protocol:

```
python3 scripts/gl_gen.py bancor 1 22 120
python3 scripts/run_prover.py gl_quad
python3 scripts/poly_gen.py bancor 1 28 120
python3 scripts/run_prover.py poly
python3 scripts/pade_gen.py bancor 1 14 120
python3 scripts/run_prover.py pade
```

(E3): For addition and multiplication constraint costs, run

```
python3 scripts/fp_add_gen.py 10000 64
python3 scripts/run_prover.py fp_add --profile-gates
python3 scripts/fp_mul_gen.py 10000 64
python3 scripts/run_prover.py fp_mul --profile-gates
```

For GeLU comparisons, run (replace 1 by 16 or 256 as in the paper)

```
python3 scripts/gl_gen.py gelu 1 24 64
python3 scripts/run_prover.py gl_quad --profile-gates
```

A.5 Notes on Reusability

Some of the microbenchmarks may report failures even though the table in the paper reports success. This can happen due to low probability events of random input values being very close to the edge of the quantization range; in such a case, that specific or set of experiments may be run again (use `python3 run_all.py --help` to see options to run only a subset of experiments).

Note that the parameters chosen for the experiments were found programmatically by a script `./scripts/find_optimal_params.py` and by increasing degree/order of the approximations (in `./scripts/run_all.py`), the failure probability can be reduced further.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Shred-to-Shine Metamorphosis of (Distributed) Polynomial Commitments

Weihan Li

A Artifact Appendix

A.1 Abstract

Polynomial commitment schemes (PCSs) enable verifying evaluations of committed polynomials. Multilinear (ML) PCSs from linear codes are valued for fast prover time. Distributed MLPCSs further shorten it by parallelizing commitment and proof generation across multiple provers.

We propose PIP$_{\text{FRI}}$, an FRI-based MLPCS that unites the fast proving of linear-time-encodable-code PCS with the compact proofs and fast verification of Reed–Solomon (RS) PCSs. PIP$_{\text{FRI}}$ has more efficient prover times than existing schemes such as Deepfold (Usenix Security '25), while remaining at least competitive proof size and verifier time.

We also propose DEPIP$_{\text{FRI}}$, the distributed version of PIP$_{\text{FRI}}$. It has full linear speedup: the prover time decreases linearly with the sub-prover number, while retaining the same proof size and verifier time with PIP$_{\text{FRI}}$. DEPIP$_{\text{FRI}}$ is also the first FRI-based distributed MLPCS supporting both a (large) single polynomnial and multiple independent polynomials.

A.2 Description & Requirements

We describe how to run our non-distributed PCS PIP$_{\text{FRI}}$ and our distributed PCS DEPIP$_{\text{FRI}}$. We then compare our PCSs with state-of-the-art schemes.

A.2.1 Security, privacy, and ethical concerns

None.

A.2.2 How to access

Our code and data are available at 10.5281/zenodo.17867141.

A.2.3 Hardware dependencies

Our non-distributed experiments require nothing special for hardwares. Our distributed experiemnts, however, needs multiple CPU cores to simulate multiple sub-provers, where one core represents one sub-prover. Note that the number of sub-provers should be power-of-two.

Our distributed benchmarks run on an Intel Xeon Platinum 8255C CPU with 24 cores. Hence, the upper bound of sub-provers in this case is 16. It is also feasible to run our experiments on AMD CPUs or CPUs with less cores. However, we recommend to use CPUs with at least 8 cores.

A.2.4 Software dependencies

Please compile and run the codes on Linux machines as we do not fully test the configurations on other platforms.

We developed and tested our codes with 32 GB RAM.

A.2.5 Benchmarks

None.

A.3 Set-up

Follow the Readme in the Zenodo Github repository to set up the required workspace. First install `Rust`. For post-installation, ensure everything is set up correctly with: `cargo --version` and `rustup --version`. Then on Linux, execute the following code to use the nightly toolchain of Rust: `rustup default nightly`.

One of our comparisons is Virgo, which uses a GKR subprotocol in Python. To run GKR (`/virgo/bench_gkr.py`), installing `python3` is required.

A.3.1 Installation

Run `cargo build`.

A.3.2 Basic Test

None.

A.4 Evaluation workflow

We only validate our paper's key results and claims, and ignore minor results or results determined by the key ones.

Specifically, for non-distributed experiments, we provide the comparison of PIP$_{\text{FRI}}$ with the state-of-the-art DeepFold.

For distributed experiments, we show that DEPIP$_{\text{FRI}}$ has full linear speedup upon PIP$_{\text{FRI}}$ and DEPIP$_{\text{FRI}}$ improves DeVirgo. The state-of-the-art distributed PCS DeVirgo is not open-sourced. To compare DEPIP$_{\text{FRI}}$ with DeVirgo, we

use the implementation of Virgo and estimate DeVirgo's performance assuming that it has full linear speedup.

Some minor results are not provided in this artifact. We do not provide other PCS comparisons (*e.g.*, Orion, PolyFRIM, mZKG and Hyrax in Figure 4) or zero-knowledge PCS comparions. These can be obtained similarly from open-sourced and our implementations, or doubly verified in the DeepFold paper. We do not provide (distributed) SNARK comparisons, which can be calculated from our (distributed) PCS benchmarks and existing (distributed) PIOP ones. We do not provide the evaluations of zero-knowledge machine learning, which can be calculated from our PCS benchmarks and other non-PCS parts from existing implementations.

A.4.1 Major Claims

(C1): Compared with DeepFold, PIP$_{\text{FRI}}$ is 5× faster in committing and 40× faster in opening, for a 14× faster prover time (committing + opening) overall. Its proof size and verifier time are competitive (around 7% worse). This is proven by the experiment (E1), whose results are reported in "**Performance of PIP$_{\text{FRI}}$.**", §6.1 and illustrated in Figure 4.

(C2): DEPIP$_{\text{FRI}}$ shows a full linear speedup over PIP$_{\text{FRI}}$. This is proven by the experiment (E2), whose results are reported in §6.2 and illustrated in Figure 6(a).

(C3): Compared with DeVirgo, DEPIP$_{\text{FRI}}$ reduces the prover time by 20×. This is proven by the experiment (E3), whose results are reported in §6.2 and illustrated in Table 6.

A.4.2 Experiments

(E1): *[Non-distributed PCS] [30 human-minutes + 30 compute minutes + 32GB memory]: Performance of* PIP$_{\text{FRI}}$ *and* DeepFold *as shown in (C1) when the polynomial size ranges from* 2^{20} *to* 2^{25} *(if the memory allows).*

Preparation of PIP$_{\text{FRI}}$: Please comment out lines 121 (`zk_commit(c, i);`), 236 (`zk_open(c, i);`), 466 (`zk_verify(c, i);`), 539-540 (`bench_single_fft, bench_multi_fft,`) in `pip_fri/benches/bench.rs`. They are not used for (C1).

Execution of PIP$_{\text{FRI}}$: Run `cargo bench -p pipfri`. It prints *type* (commit, open, and verify), *variable* (from 20 to 25, corresponding to polynomial size 2^{20} to 2^{25}), and the estimated times (the bold one of three). It also prints the proof size when running the verify part.

Results of PIP$_{\text{FRI}}$: Collect the commit, open, verify times and the proof size of PIP$_{\text{FRI}}$. In our experiments, they are shown in Table 1. Our claim (C1) directly follows from the table.

Note that apart from running times, the obtained proof size may also slightly differ from the table, as the proof includes Merkle-tree multi-openings, which are random.

Also, in Table 1, the opening time of PIP$_{\text{FRI}}$ when the polynomial size is 2^{24} does not double over that when the size is 2^{23}. This is because the adjustable parameter of our shred-to-shine framework (m and ℓ) changes, as shown in **Parameters**, §6.1.

Table 1: Results of (E1) in our experiments

Scheme	Size	Commit (s)	Open (ms)	Verify (ms)	Proof (KB)
PIP$_{\text{FRI}}$	2^{20}	0.55	0.23	1.56	156
	2^{21}	1.13	0.46	1.69	176
	2^{22}	2.34	0.94	1.87	196
	2^{23}	4.80	1.85	2.08	228
	2^{24}	9.29	1.89	2.27	262
	2^{25}	19.11	3.69	2.38	288
DeepFold	2^{20}	2.85	8.19	1.50	153
	2^{21}	5.83	16.67	1.58	169
	2^{22}	11.57	33.79	1.84	194
	2^{23}	23.79	69.76	1.90	213

Preparation of DeepFold: Given a 32GB RAM, DeepFold can only run when the polynomial size is no more than 2^{23}. Hence, modify the lines 11 from `const SIZE: usize = 25;` to `const SIZE: usize = 23;` in `deepfold/benches/bench.rs`.

Execution of DeepFold: Run `cargo bench -p deepfold`. It prints *type* (commit, open, and verify), *variable* (from 20 to 23, corresponding to polynomial size 2^{20} to 2^{23}), and the estimated times (the bold one of three). It also prints the proof size when running the verify part.

Results of DeepFold: Collect the commit, open, verify times and the proof size of PIP$_{\text{FRI}}$. In our experiments, they are shown in Table 1. Note that apart from running times, the obtained proof size may also slightly differ from the table, as the proof includes Merkle-tree multi-openings, which are random.

Results of (E1): As shown in Table 1, compared with DeepFold, PIP$_{\text{FRI}}$ has around 5× faster committing time, 35-37× faster opening time. The verifier time and proof size are competitive. This verifies our major claim (C1).

(E2): *[Distributed PCS with full linear speedup] [30 human minutes + 30 compute minutes + 32 GB memory + ≥8 cores]:* DEPIP$_{\text{FRI}}$*'s full linear speedup as shown in (C2) when the sub-prover number ranges from* 2 *to* 8*. For simiplicity, we fix the polynomial size to be* 2^{20}.

Preparation of DEPIP$_{\text{FRI}}$: Modify the line 46 as `let variable_num: usize = 20;` in `de_pip_fri/examples/de_pip_fri.rs`. This fixes the polynomial size to be 2^{20}.

Execution of DEPIP$_{\text{FRI}}$: Run `cd de_pip_fri`. Run `./run_benchmark.sh <sub-prover number> <running times>`. Set `running time` = 10. Set `sub-prover number` to be 2, 4, or 8.

Results of DePip$_{\text{FRI}}$: Collect the commit, open, verify times and the proof size of DePip$_{\text{FRI}}$ from `Process 0`, *i.e.*, the master prover. In our experiments, they are shown in Table 2. Note that apart from running times, the obtained proof size may also slightly differ from the table, as the proof includes Merkle-tree multi-openings, which are random. Also note that our paper run non-distributed and distributed experiments on two machines (as shown in §6), so the performances of Pip$_{\text{FRI}}$ in the two tables are different.

As shown, with the increase of the sub-prover number, the committing and opening times decreaes linearly. Further, the verifier time and proof size do not change greatly. This verifies our major claim (C2).

Table 2: Results of (E2) and (E3) in our experiments

Scheme	#Prover	Commit (s)	Open (s)	Verify (ms)	Proof (KB)
Pip$_{\text{FRI}}$	1	0.88	0.33	2.18	156
	2	0.44	0.17	2.32	156
DePip$_{\text{FRI}}$	4	0.22	0.087	2.31	154
	8	0.11	0.047	2.32	154
Virgo	1	2.95	18.6 + 10.8	1.47 + 0.27	169 + 51
	2	1.47	14.7	1.74	220
DeVirgo	4	0.74	7.4	1.74	220
	8	0.37	3.7	1.74	220

(E3): *[Distributed PCS comparison] [30 human minutes + 30 compute minutes + 32 GB memory + ≥8 cores]: Performance of* DePip$_{\text{FRI}}$ *and* DeVirgo *as shown in (C3) when the sub-prover number ranges from 2 to 8 and the polynomial size is* 2^{20}.

Preparation, execution and results of Virgo: We estimate the performance of non-open-sourced DeVirgo from its non-distributed scheme Virgo.

First, modify lines 21-22 as `const SMALL: usize = 20; const SIZE: usize = 20;` in `virgo/benches/bench.rs`. This fixes the polynomial size to be 2^{20}.

Second, comment out lines 96 (`zk_commit(c, i);`), 204 (`zk_open(c, i);`), and 336 (`zk_verify(c, i);`) in `virgo/benches/bench.rs`. This is because we do not need the zk version here.

Third, run `cargo bench -p virgo` to obtain the **the first-part** performance of the Virgo PCS. It prints *type* (commit, open, and verify) and the estimated times (the bold one of three). It also prints the proof size when running the verify part.

Fourth, obtain **the second-part** performance of the Virgo PCS. Modify line 9 as `for i in range(20, 21):` in `virgo/bench_gkr.py`. This fixes the polynomial size to be 2^{20}. Run `python3 bench_gkr.py`. It would print the prover time (working during the open phase), verifier time, and the proof size.

Finally, compute the performance of Virgo from the two parts above. The results of our experiments is shown in Table 2. For columns `Open, Verifiy, Proof`, the first term comes from the first part, and the second term comes from the second part.

Results of DeVirgo**:** Estimate the performance of DeVirgo assuming that it has full linear speedup. The results is shown in Table 2. As shown, fixing the sub-prover number, the prover time of DePip$_{\text{FRI}}$ is $> 20\times$ faster than DeVirgo. This verifies our major claim (C3).

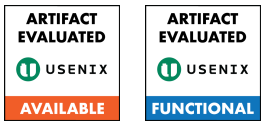
Notes on communication: Our experiments of DePip$_{\text{FRI}}$ prints the communication for each prover. To compare with the unopen-sourced DeVirgo, we estimate its communication using our implementations. Specifically, using lines 104-105 `Net::stats().bytes_recv` and `Net::stats().bytes_sent` helps to record the bytes the current sub-prover recieves or sends. Hence, it is feasible to record the concrete communication overhead of the `de_prover.de_commit_polynomial()` in line 164. Our distributed commitment algorithm is basically the same as DeVirgo for both using the committing algorithm of distributed FRI. Thus we can estimate DeVirgo's communication overhead of distributed commitment using our algorithm. Besides, the communication of DeVirgo's opening algorithm is basically the same as its commitment algorithm. Hence we estimate DeVirgo's total communication by multiplying its commitment overhead by two.

A.5 Notes on Reusability

We implement Pip$_{\text{FRI}}$ using our shred-to-shine framework Pip and an existing PCS PolyFRIM. The parameters m (*i.e.*, the size of shredded polynomials) are adjustable for the trade-offs between the running times and the proof size. Our framework is also promising to align with other code-based PCSs like WHIR and Basefold. This is left for the 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Quantifying Large Language Model Attacks Through the Lens of Model Cognition

Xiuming Liu[1]*, Chaoxiang He[1]*, Xuanran Yu[1], Jichen Chai[1], Feiyue Xu[1], Sheng Hang[1], Hanqing Hu[1], Bin Benjamin Zhu[2], Hongsheng Hu[1], Shi-Feng Sun[1], Dawu Gu[1], Shuo Wang[1]†

[1]Shanghai Jiao Tong University, Shanghai, China
[2]Microsoft Corporation, China

A Artifact Appendix

A.1 Abstract

This artifact contains the source code, pre-processed datasets, and evaluation scripts required to reproduce the key findings presented in our paper. The artifact includes our implementation of the layer-wise toxicity probing framework, the multi-layer sentinel defense, and the cognitive drift analysis. Evaluators can reproduce the training of probes, the construction of the multi-layer sentinel, and the comparative evaluation against baselines (e.g., Llama-Guard-3-8B) on adversarial benchmarks (Sneaky, I2P, MMA, Labelled) using the provided Python scripts. The artifact supports reproducibility on standard GPUs (e.g., NVIDIA A100) and facilitates further research into internal model cognition and safety.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact is low-risk, relying on standard open-source libraries without requiring root privileges or altering system security settings. It uses pre-filtered text datasets free of explicit imagery or personally identifiable information and runs entirely on local, open-weight models, preventing data transmission to external APIs. While designed for defensive research, the evaluation process involves handling potentially offensive text strings.

A.2.2 How to access

The artifact is available on Zenodo: https://doi.org/10.5281/zenodo.17875176.

A.2.3 Hardware dependencies

A GPU is required to run this artifact. We recommend using an environment with a single NVIDIA A100 GPU (40GB VRAM) or equivalent to comfortably run experiments on models up to 8B parameters. Approximately 35GB of disk space is needed to store the target language models, datasets, and intermediate files generated during execution. The experiments reported in the paper were primarily conducted on a single NVIDIA A100 GPU (80GB VRAM).

A.2.4 Software dependencies

All experiments were conducted in a Conda environment with Python 3.10.19 on Ubuntu 22.04. The artifact relies on standard open-source Python libraries. All necessary dependencies can be installed via the commands in the Jupyter notebook. Detailed installation instructions are provided in Section A.3.1.

A.2.5 Benchmarks

The artifact relies on several publicly available models and datasets essential for reproducing the experiments reported in the paper.

- **Models:** The primary target models used for evaluation are open-weight LLMs from the Qwen (e.g., Qwen3-4B) and LLaMA families, which are automatically downloaded from the Hugging Face Hub. For baseline comparisons, the artifact also utilizes "Llama-Guard-3-8B" and "NSFW_text_classifier". **Note** that LLaMA models are gated and require explicit agreement to Meta's Community License.
- **Datasets:** We provide scripts and instructions to reconstruct the training and evaluation datasets.
  - **Training Data:** A set of 2,000 prompts (1,000 toxic, 1,000 benign) constructed from filtered NSFW-56k captions and GPT-4o generated "home scene" descriptions.
  - **Evaluation Benchmarks:** We evaluate on four standard adversarial datasets: **I2P**, **Sneaky**, **MMA**,

*The authors contribute equally to this paper.
†Corresponding authors.

and **Labelled**. These datasets cover diverse categories of harmful content including hate speech, self-harm, and explicit material.

A.3 Set-up

A.3.1 Installation

We recommend using Conda. Open the **src/installation.ipynb** notebook and execute the cells sequentially to set up the environment.

First, the notebook will create and prepare a Python 3.10 environment named 'lac'. After creating the environment, you will need to activate it within your Jupyter kernel or select the 'lac' kernel. Then, run the subsequent cells to install dependencies and download the necessary models directly within the notebook.

Note: The notebook is pre-configured to download `Qwen3-4B`. If you wish to use a different model, locate the cell with the **TODO** comment and modify the 'model_ids' list before execution.

A.3.2 Basic Test

To verify the setup, first edit **src/basic_test.py** to uncomment your downloaded model (e.g., `Qwen3-4B`) in the **TODO** section. Then, execute:

```
conda activate lac
python src/basic_test.py
```

Expected Output: The script checks GPU availability and dependencies. If successful, it will output:**Ready for reproduction!**

A.4 Evaluation workflow

A.4.1 Major Claims

Our paper makes the following major claims:

(C1): *Layer-wise Separability of Toxic Intent (O1).* Intermediate hidden states in LLMs separate harmful from benign prompts with high accuracy (AUC ≥ 0.90) in mid-depth layers, even when the model ultimately generates a safe response. This is proven by experiment (E1) described in Section 3.3 and Figure 4 of the paper.

(C2): *Cognitive Drift under Perturbation (O2 & O3).* Adversarial perturbations cause significant divergence in hidden state representations (cognitive drift), particularly in early and late layers, while mid-layers remain relatively stable. The degree of drift correlates with attack success. This is proven by experiment (E2) described in Section 4.1.2 and Figure 5 of the paper.

(C3): *Effectiveness of Multi-Layer Sentinel (O4).* A lightweight multi-layer sentinel, constructed by fusing non-redundant layer probes, achieves superior detection accuracy on adversarial benchmarks (Sneaky, I2P, MMA, Labelled) compared to single-layer probes and external baselines. This is proven by experiment (E3) described in Section 4.1.4, Table 3 and Figure 7 of the paper.

(C4): *Superiority over Baselines.* The multi-layer sentinel outperforms existing safety mechanisms, including generative judgment (LLM Judge), NSFW_text_classifier, and Llama-Guard-3-8B, particularly on stealthy attacks like I2P and heterogeneous real-world data (Labelled dataset). This is proven by experiment (E4) described in Section 4.1.4 and Table 3 of the paper.

A.4.2 Experiments

The following experiments provide the operational steps to reproduce the evidence for the major claims (C1–C4) listed above. To facilitate modular verification, we have provided four dedicated Jupyter Notebooks located in the `src/claims/` directory (i.e., `claim1.ipynb` through `claim4.ipynb`), where each notebook corresponds directly to a specific claim. This artifact demonstrates the full workflow using **Qwen3-4B** as the representative model. All experiments are configured to run successfully on a single NVIDIA A100 GPU.

(E1): *[Layer-wise Probe Training] [10 human-minutes + 30 compute-minutes + 2GB disk]: Quantifies the linear separability of toxic intent at each layer.*

How to: Open and run `src/claims/claim1.ipynb`. This notebook implements the hidden state extraction and MLP probe training logic to quantify layer-wise separability.

Preparation: Ensure the environment is correctly set up and the `Qwen3-4B` model has been downloaded via `installation.ipynb`.

Execution: Execute all cells in the notebook. The code first extracts hidden states for the training set (P_{harm} and P_{benign}) from all 36 layers and saves them as `.pt` files. It then trains a lightweight MLP classifier for each layer.

Results: The notebook outputs a table of performance metrics (Accuracy, F1, AUC, TPR@FPR) for each layer's probe. The results should reproduce the findings in Figure 4, demonstrating high separability in the middle layers (e.g., layers 10–25) compared to early layers.

(E2): *[Cognitive Drift Analysis] [5 human-minutes + 60 compute-minutes]: Analyzes how adversarial perturbations affect layer-wise toxicity detection.*

How to: Open and run `src/claims/claim2.ipynb`. This notebook applies the trained single-layer probes to perturbed prompts to calculate detection rates across layers.

Preparation: Requires the trained layer-wise probes from (E1).

Execution: Execute all cells in the notebook. The code

first loads the `fixed.json` dataset (containing original and perturbed prompts) and applies the trained probes to extract layer-wise predictions, saving them to `processed_predictions.json`. It then aggregates these predictions to calculate toxicity rates for each perturbation strategy.

Results: The notebook generates and displays a heatmap PDF (`qwen_4b_combined_heatmap.pdf`). This reproduces the Qwen3-4B subplot in Figure 6, visualizing how different perturbation types (e.g., prefix vs. suffix) cause varied detection sensitivity across the model's layers.

(E3): *[Sentinel Construction & Evaluation] [5 human-minutes + 20 compute-minutes]: Trains the multi-layer sentinel and evaluates it on adversarial benchmarks.*

How to: Open and run `src/claims/claim3.ipynb`. This notebook performs layer selection, trains the fused multi-layer sentinel, and evaluates its performance on both adversarial benchmarks and the Labelled dataset.

Preparation: Requires the trained layer-wise probes from (E1).

Execution: Execute all cells in the notebook. The code first analyzes layer-wise correlations to select non-redundant layers (e.g., layers [3, 4, 9, 12, 19, 24, 31]). It then trains the sentinel on these selected layers and evaluates it against the `Sneaky`, `I2P`, `MMA`, and `Labelled` datasets.

Results: The notebook prints classification metrics (Accuracy, Precision, Recall, F1, AUC, TPR@FPR) for the Multi-Layer Sentinel. These numbers should closely match the "Qwen3-4B Multi-layer" rows in Table 3 (e.g., Sneaky Accuracy ≈ 0.9320, I2P Accuracy ≈ 0.8863).

(E4): *[Baseline Comparison] [5 human-minutes + 60 compute-minutes]: Evaluates external baselines on the same benchmarks.*

How to: Open and run `src/claims/claim4.ipynb`. This notebook evaluates the performance of the `Generative Judgment (LLM Judge)` alongside external safety models (`NSFW_text_classifier` and `Llama-Guard-3-8B`).

Preparation: Ensure that the baseline models have been downloaded to the `models/` directory via `installation.ipynb`.

Execution: Execute all cells in the notebook. The code first evaluates the `Generative Judgment` baseline, where the model itself is prompted to score the safety of inputs. It then proceeds to evaluate the `NSFW_text_classifier` pipeline and the `Llama-Guard-3-8B` model against the `Sneaky`, `I2P`, `MMA`, and `Labelled` datasets.

Results: The notebook outputs detailed performance tables (Accuracy, F1, AUC, TPR@FPR) for all three baselines. Comparing these results with the Multi-Layer Sentinel metrics from (E3) validates Claim (C4).

(E5): *[Instant Result Verification] [< 1 human-minute + < 1 compute-minute]: Instantly verify specific experimental results reported in the paper (e.g., Table 2) using pre-computed logs.*

How to: Use the provided `src/quick_start.py` CLI tool to query the `results/metrics.json` file. This script parses the finalized experimental data and allows evaluators to retrieve exact metrics for any Model-Method-Dataset combination reported in the paper.

Preparation: Ensure the `lac` environment is activated. No GPU is required for this step.

Execution: Run the following command to retrieve the performance metrics for the **Multi-layer** sentinel on **Qwen3-4B** against the **Sneaky** dataset:

```
conda activate lac
python src/quick_start.py --model
  ↪ Qwen3-4B --method Multi-layer
  ↪ --dataset Sneaky
```

Note: You can run

```
python src/quick_start.py --help
```

to view all available models, methods, and datasets.

Results: The script will print a formatted list of metrics, including Accuracy, Precision, Recall, F1, and AUC. The output values should match the corresponding entry in **Table 3** of the paper. You can modify the `-dataset` argument (e.g., to `I2P` or `MMA`) to verify other rows in the table.

A.5 Notes on Reusability

This artifact provides a modular, architecture-agnostic framework suitable for practical deployment and extended research.

Real-Time Dynamic Truncation. The lightweight multi-layer sentinel enables real-time internal content moderation with negligible latency ($\approx$ 5ms). By monitoring hidden states, systems can trigger an "early exit" upon detecting harmful intent, preventing the generation of toxic content and significantly reducing inference costs compared to post-hoc filtering.

Extensibility. While focused on toxicity, the codebase is label-agnostic. Researchers can easily adapt the data loading scripts to train probes for other internal states by simply swapping the training dataset.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Certified in Theory, Broken in Practice: Assumption Gaps in Cryptographic Model Certification

Carter Luck, Olive Franzese-McLaughlin, Elisaweta Masserova,
Akira Takahashi, Antigoni Polychroniadou, Nicolas Papernot

A Artifact Appendix

A.1 Abstract

This artifact contains the scripts used to reproduce the data-forging attacks described in the paper. The attacks target three types of ML model auditing schemes—decision trees, neural networks, and XGBoost ensembles—and demonstrate that a malicious model trainer can craft a training set that causes the model to behave correctly on the audited sample while performing poorly (or unfairly) on the general population. The artifact supports reproduction of Figures 3–8.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

None. The artifact runs self-contained ML experiments on public datasets and does not require network access, elevated privileges, or interaction with any external service.

A.2.2 How to access

The artifact is available at: https://doi.org/10.5281/zenodo.20337317

A.2.3 Hardware dependencies

No special hardware is required. All experiments were run on a standard CPU machine. We tested experiments on Ubuntu 24.04 with 13th Gen Intel(R) Core(TM) i5-1335U and 16 GB RAM.

A.2.4 Software dependencies

- Python 3.12
- conda (recommended) or pip
- Key packages: `xgboost 3.0.4`, `scikit-learn 1.7.1`, `torch 2.6.0`, `numpy 2.4.4`, `pandas 2.3.1`, `scipy 1.16.0`, `matplotlib 3.10.3`, `folktables 0.0.12`, `aif360`, `ucimlrepo`

All dependencies are specified in `environment.yml` (conda) and `requirements.txt` (pip).

A.2.5 Benchmarks

Three public tabular datasets are used:

- **Adult** (UCI) — income prediction, 48 842 records, 11 features. Included as `data/adult.csv`.
- **Default Credit** (UCI) — credit card default prediction, 30 000 records, 23 features. Included as `data/default_credit.csv`.
- **ACS Income** (folktables) — income prediction from the American Community Survey, 10 features. Included as `data/acs_income.csv`. Run `python data_download.py` to download the raw ACS data if needed.

A.3 Set-up

A.3.1 Installation

1. Create and activate the conda environment (recommended):

   ```
   conda env create -f environment.yml
   conda activate adversarial-arborists
   ```

   Alternatively, install via pip (Python 3.12 required):

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

A.3.2 Basic Test

Run a quick smoke test with reduced parameters to verify the installation:

```
python xgboost_attacks.py \
  --n-estimators 50 --n-trials 1
```

Expected output (values will vary slightly):

```
Loading data/default_credit.csv ...
  30000 rows, 23 features, rate=0.221
  XGBoost: 50 trees, depth 10
  trial 0: audit=0.XXXX eval=0.XXXX
Results over 1 trials (xgboost):
  S_audit: 0.XXXX +- 0.0000
  S_eval : 0.XXXX +- 0.0000
```

The key indicator of a successful run is that `S_audit` accuracy is substantially higher than `S_eval` accuracy.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): A malicious trainer can craft S'_{train} such that a decision-tree model achieves near-perfect accuracy on the audit set S_{audit} while collapsing to near-random accuracy on the held-out evaluation set S_{eval}. This is shown in experiments (E1), (E2), and (E3), whose results are illustrated in Figures 3–6, produced by `*_evasion.py`.

(C2): The same attack generalizes to XGBoost ensembles. As the attack parameter p increases from 0 to 1, audit accuracy rises toward 1.0 while real-world accuracy falls. This is proven by the experiment (E4) described in Appendix D.1 whose results are illustrated in Figure 7, produced by `xgboost_attacks.py --plot`.

(C3): The same attack generalizes to neural networks. This is proven by the experiment (E5) described in Appendix D.2 whose results are illustrated in Figure 8 (neural network variant), produced by `nn_attacks.py`.

A.4.2 Experiments

(E1): Decision-tree attacks (Figures 3 and 5) [1 human-minute + 2 compute-hours + 0GB disk]: Run the per-dataset evasion scripts on their default settings.
Preparation: Ensure the conda environment is active. Download ACS data if needed:

```
python data_download.py
```

Execution:

```
python folktables_evasion.py \
  -v --csv ACS
python adult_evasion.py \
  -v --csv Adult
python credit_evasion.py \
  -v --csv Credit
python compas_evasion.py \
  -v --csv COMPAS
python german_evasion.py \
  -v --csv German
python crime_evasion.py \
  -v --csv Crime
python make_graph.py \
  --path data/ --target acc \
  --files ACS Adult COMPAS
python make_graph.py \
  --path data/ --target acc \
  --files German Credit Crime
```

Output can be saved to a file with `-o <file>`.
Results: Each of the first six scripts print accuracy on S_{audit} and S_{eval} across trials, along with fairness metrics. The values correspond to the rows of Table 2 and the data points in Figures 3 and 5. Audit accuracy should be substantially higher than evaluation accuracy. The last two scripts display three-panel figures. The first should match Figure 3 in the paper and the second should match Figure 5: as p increases, the orange (Audit Performance) curve rises towards 1.0 and the blue (Real-World Performance) curve falls.

(E2): Decision-tree fairness attacks (Figure 4) [1 human-minute + 1 compute-hour + 0GB disk]: Run the per-dataset evasion scripts on fairness.
Preparation: Ensure the conda environment is active. Download ACS data if needed:

```
python data_download.py
```

Execution:

```
python folktables_evasion.py \
  -v --csv ACS --target fair
python adult_evasion.py \
  -v --csv Adult --target fair
python compas_evasion.py \
  -v --csv COMPAS --target fair
python make_graph.py \
  --path data/ --target fair \
  --files ACS Adult COMPAS
```

Output can be saved to a file with `-o <file>`.
Results: Each of the first three scripts print accuracy and fairness metrics on S_{audit} and S_{eval} across trials. The values correspond to the data points in Figure 4. Audit accuracy should be substantially higher than evaluation accuracy. The last script displays a three-panel figure. It should match Figure 4 in the paper: as p increases, the orange (Audit Fairness) curve rises towards 1.0 and the blue (Real-World Fairness) curve falls.

(E3): Decision-tree denial attack (Figure 6) [1 human-minute + 20 compute-minutes + 0GB disk]: Run the per-dataset evasion scripts on denial-rate.
Preparation: Ensure the conda environment is active. Download ACS data if needed:

```
python data_download.py
```

Execution:

```
python folktables_evasion.py \
  -v --csv ACS --target zero
python make_graph.py \
  --path data/ --target zero \
  --files ACS
python make_graph.py \
  --path data/ --target acc \
  --files ACS
```

Output can be saved to a file with `-o <file>`.
Results: The first three script prints accuracy and denial rate on S_{audit} and S_{eval} across trials. The values correspond to the data points in Figure 6. Audit accuracy should be substantially higher than evaluation accuracy, and evaluation denial rate should be higher than audit denial rate. The last two scripts each display a figure. They should match Figure 6's left and right graphs respectively: as p increases, the audit denial rate should stay near its original value, bending upwards slightly

near $p = 0.5$, while the real-world denial rate rises towards 1.0; meanwhile, real-world accuracy increases towards 1.0 while the audit accuracy slightly decreases as p increases.

(E4): XGBoost attack — Figure 7 [1 human-minute + 10 compute-minutes + 0GB disk]: Reproduce the p-sweep plot showing Audit Performance vs. Real-World Performance across all three datasets.

Preparation: Ensure the conda environment is active.

Execution:
```
python xgboost_attacks.py \
  --plot --output figure7.png
```
This sweeps $p \in \{0.0, 0.1, \ldots, 1.0\}$, runs 5 trials per value, and saves a three-panel figure to `figure7.png`. Runtime is approximately 10 minutes on a standard CPU.

Results: `figure7.png` should match Figure 7 in the paper: as p increases, the orange (Audit Performance) curve rises toward 1.0 and the blue (Real-World Performance) curve falls, with narrow $\pm$std shading across all three datasets.

(E5): Neural-network attack [1 human-minute + 8 compute-hours + 3GB disk]: Reproduce the neural-network variant of the attack.

Preparation: Ensure the conda environment is active.

Execution:
```
python nn_attacks.py --sweep
```
This sweeps over the size of S'_{train} by adding more copies of S_{audit}, charting the accuracy of the resultant models. Runtime may be up to several hours, as this involves training 10 large neural networks.

Results: The displayed chart should match Figure 8 in the paper: the blue curve (Audit Performance) stays relatively constant, while the orange curve (Real-World Performance) is consistently higher, growing to 1 as the x-axis increases.

A.5 Notes on Reusability

The attack scripts accept command-line arguments that allow easy experimentation beyond the paper's settings: See the `README.md` in the repository for a full description of all options for each script.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: MATE: Policy-Aware Security Auditing for Mobile Agents via Synthesis-Driven Trajectory Learning

Changyue Jiang
Fudan University
Shanghai Innovation Institute
cyjiang24@m.fudan.edu.cn

Jiayi Wang
Fudan University
24212010031@m.fudan.edu.cn

Xin Wen
Fudan University
25213050398@m.fudan.edu.cn

Jiarun Dai
Fudan University
jrdai@fudan.edu.cn

Geng Hong
Fudan University
ghong@fudan.edu.cn

Xudong Pan
Fudan University
Shanghai Innovation Institute
xdpan@fudan.edu.cn

A Artifact Appendix

A.1 Abstract

This artifact supports the availability and functionality evaluation of *#2172* MATE*: Policy-Aware Security Auditing for Mobile Agents via Synthesis-Driven Trajectory Learning*. MATE is a lightweight, policy-conditioned auditor for mobile-agent trajectories: given a user instruction, an agent trajectory, and a natural-language security policy, it predicts whether the trajectory violates the policy and produces an explanatory rationale. The artifact contains the code, MATEBENCH benchmark data, trained MATE-0.5/1.5/3B model checkpoints, policy retriever, data synthesis and augmentation pipeline, trajectory adapter, recorded evaluation outputs, and documentation promised in the paper's Open-Science section.

The artifact supports availability, functionality, and results-reproduction evaluation. It provides a lightweight functionality workflow that starts a local MATE model server, builds a small MATEBench subset, runs the evaluator, and summarizes the resulting metrics. It also includes the released checkpoints, complete benchmark splits, full-evaluation commands, and recorded outputs for verifying the paper's primary MATEBench results. Full-benchmark evaluation requires additional GPU time and model-serving resources, so the appendix distinguishes lightweight functionality checks from full-scale reproduction workflows. (**Readers are encouraged to refer to the `README.md` file in the `MATE-Artifacts.zip` archive and follow its detailed instructions when performing validation.**)

A.2 Description & Requirements

A.2.1 Security, Privacy, and Ethical Concerns

The archive contains no API keys, tokens, real user accounts, or private credentials. The real-world MATEBENCH subset is released as sanitized structured trajectory data and textual screen summaries rather than private raw phone logs or unsanitized screenshots. Some benchmark examples describe security-relevant actions, such as credential handling or unsafe sharing, but they are synthetic or sanitized evaluation records rather than instructions for attacking real systems. Reviewers should run experiments in an isolated environment and supply their own model-serving credentials if they choose to use external LLM endpoints for synthesis or adapter workflows.

A.2.2 How to Access

The artifact is publicly available on Zenodo at https://doi.org/10.5281/zenodo.20232924. This DOI serves as the authoritative and stable access point for the artifact.

The Zenodo record provides an archive named `MATE-Artifacts.zip`. After downloading and extracting the archive, reviewers should begin with `README.md`, `ARTIFACT_MANIFEST.md`, and `claims_mapping.md`. These documents describe the scope and organization of the artifact, map the released components to the claims made in the paper, and provide the required setup and evaluation commands.

The archive is organized as follows. `MateBench/` contains the released benchmark datasets. `Mate_model/` contains the MATE-0.5B, MATE-1.5B, and MATE-3B checkpoints. `evaluation/` contains the evaluator, the vLLM launcher, and the recorded evaluation outputs. `policy_retriever/` contains the retriever model and the security-policy pools. `data_synthesis_pipeline/`,

trajectory_adapter/, and model_deployment/ contain the data-synthesis, trajectory-adaptation, and deployment utilities described in the paper.

A.2.3 Hardware Dependencies

Functionality and reproducibility evaluations require serving the released MATE-0.5B, MATE-1.5B, or MATE-3B checkpoint through vLLM or another OpenAI-compatible inference server. We recommend a Linux machine equipped with at least one NVIDIA H100 GPU. This configuration can serve all three released checkpoints and supports both the minimal subset test and full evaluation of the MATEBENCH splits. Reviewers may use the smaller MATE-0.5B or MATE-1.5B checkpoint for a faster functionality check. Full-benchmark evaluation requires additional runtime, and regenerating the synthetic data also requires access to an external or locally served LLM endpoint.

The largest directories are Mate_model/ (approximately 9.6GB), policy_retriever/ (approximately 2.1GB), and evaluation/ (approximately 2.5GB, including recorded outputs and the local rationale-embedding model). The MATEBENCH benchmark itself is approximately 21MB.

A.2.4 Software Dependencies

The artifact was prepared for Python 3.11. The base Python dependencies are listed in requirements.txt. vLLM is used only for local model serving and can be installed separately if reviewers run the minimal functionality test.

```
conda create -n mate python=3.11 -y
conda activate mate
pip install -r requirements.txt
pip install vllm
```

The evaluator uses an OpenAI-compatible chat-completions endpoint. The scripts set MATE_MODEL_NAME, MATE_BASE_URL, and MATE_API_KEY automatically for the minimal workflow. Data synthesis and trajectory-adapter scripts have their own endpoint configurations and may require external LLM services if reviewers regenerate data.

A.2.5 Benchmarks

MateBench/ contains 4,087 JSON samples: 2,775 MATEBENCH-IN samples, 1,150 MATEBENCH-OUT samples, and 162 MATEBENCH-REAL sanitized real-world samples. Each JSON file contains a user instruction, trajectory steps, a natural-language security policy, and a reference Violation/Category/Rationale label. The in-domain and out-of-domain splits cover synthetic trajectory-level auditing tasks, while MateBench-Real/ contains manually collected, sanitized mobile-agent trajectories.

A.3 Set-up

A.3.1 Installation

Unpack MATE-Artifacts.zip, enter the artifact directory, and install the base dependencies:

```
cd MATE-Artifacts
conda create -n mate python=3.11 -y
conda activate mate
pip install -r requirements.txt
```

A.3.2 Basic Test

Reviewers can verify the benchmark size and summarize recorded outputs:

```
find MateBench -name '*.json' | wc -l
python summarize_results.py \
  --root evaluation/evaluation_results
python summarize_results.py \
  --root evaluation/
    evaluation_results_subset
```

The expected MATEBENCH count is 4,087 JSON files. The summary commands should print Markdown tables with task, model, run identifier, sample count, accuracy, violation F1, rationale similarity, precision, and recall. The subset-result directory contains six MATE-1.5B runs generated by following the README workflow.

A.4 Evaluation Workflow

A.4.1 Major Claims

The artifact supports the following claims from the paper and Open-Science section.

1. The MATE auditing and evaluation code is released. Evidence: evaluation/, run_minimal_demo.sh, and run_eval_subset.sh.
2. MATEBENCH is released with synthetic in-domain, synthetic out-of-domain, and sanitized real-world subsets. Evidence: MateBench/.
3. Trained MATE checkpoints are released. Evidence: Mate_model/.
4. The policy retriever, policy pools, data synthesis pipeline, augmentation pipeline, and trajectory adapter are released. Evidence: policy_retriever/, data_synthesis_pipeline/, trajectory_adapter/, and model_deployment/.

5. Recorded outputs for the reported evaluation style are included. Evidence: `evaluation/evaluation_results/` and `evaluation/evaluation_results_subset/`.

A.4.2 Experiments

Experiment 1: availability inspection. Inspect the three primary reviewer-entry documents and verify the presence of the major artifact directories:

```
ls README.md ARTIFACT_MANIFEST.md
    claims_mapping.md
ls MateBench Mate_model policy_retriever
    evaluation
```

This experiment confirms that the promised code, datasets, models, policies, and recorded outputs are present in the archive for reviewer inspection.

Experiment 2: recorded-result summary. Run:

```
python summarize_results.py --root
    evaluation/evaluation_results
python summarize_results.py --root
    evaluation/evaluation_results_subset
```

The expected output is a compact Markdown table. The full recorded tree contains 18 run directories, and the subset tree contains 6 MATE-1.5B subset run directories. This experiment lets reviewers inspect the result format and metrics without rerunning model inference.

Experiment 3: subset construction without model serving. Run:

```
CREATE_ONLY=1 SUBSET_SIZE=2 MATE_TASK=in
    MATE_LANG=en \
  bash run_eval_subset.sh
```

The expected output is a temporary directory containing sampled MATEBench-In English files from the trajectory, multi-app, multi-policy, and no-match subdirectories. This is a lightweight check of dataset layout, sampling logic, and script behavior.

Experiment 4: minimal end-to-end functionality. On a machine capable of serving MATE-1.5B, run:

```
bash run_minimal_demo.sh
```

The script starts vLLM, waits for `/v1/models`, builds a temporary MATEBENCH-IN English subset, runs `evaluation/run_matebench.py`, and summarizes the result. The default configuration is `MODEL_PATH=Mate_model/MATE-1.5B`, `SERVED_MODEL=MATE-1.5B`, `PORT=8016`, `MATE_TASK=in`, `MATE_LANG=en`, `SUBSET_SIZE=50`, and `BATCH_SIZE=8`. The expected output is a new directory under `evaluation/evaluation_results_subset/` with `eval_results.txt` and `eval_model_output.json`.

If reviewers already have an OpenAI-compatible model endpoint running at `http://127.0.0.1:8016/v1`, they can skip server startup:

```
START_SERVER=0 MODEL_NAME=MATE-1.5B PORT
    =8016 \
  MATE_TASK=in MATE_LANG=en bash
    run_eval_subset.sh
```

If every sample reports `API error: Connection error`, the model endpoint is not reachable. First check:

```
curl http://127.0.0.1:8016/v1/models
```

In local runs, clients should normally connect through `127.0.0.1` or the server's real IP address. `0.0.0.0` is appropriate as a server bind address, but it is not the recommended client address.

A.5 Notes on Reusability

The artifact supports reuse beyond availability evaluation. Researchers can evaluate OpenAI-compatible auditors on MATEBENCH by serving a model endpoint and specifying its name in `evaluation/run_matebench.py`. The policy retriever and policy pools can attach app-specific security policies to new mobile-agent trajectories, while the trajectory adapter and deployment utilities convert raw agent logs into the canonical MATE format. The synthesis and augmentation code can generate policy-conditioned trajectories, although exact regeneration depends on the external LLM endpoint, model version, decoding settings, and API availability.

The release excludes API keys, real user accounts, private raw phone logs, and unsanitized screenshots. The real-world subset contains sanitized structured trajectories, preserving research utility for inspection and benchmarking while reducing privacy and platform risks.

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/usenixsec2026/.

USENIX Security'26 Artifact Appendix: When Address Learning Goes Wrong: Inducing Forwarding Loops and DoS Amplification in SDN

Dezhang Kong[1], Yilun Zhang[2], Zekun Xie[3], Ningpeng Zheng[4], Shi Lin[5], Zhenhua Xu[1], Minghao Li[6]
Zhebo Wang[1], Xiang Chen[1], Changting Lin[1], Dong Zhang[7], Xuan Liu[8], Chunming Wu[1], Meng Han[1]
[1]Zhejiang University, [2]Purdue University, [3]Xi'an University of Posts & Telecommunications
[4]Yancheng Institute of Technology, [5]Zhejiang Gongshang University, [6]Chongqing University
[7]Fuzhou University, [8]Yangzhou University

A Artifact Appendix

A.1 Abstract

Our artifact contains the **source code, packet traces, raw data, and plotting material** used for the paper "When Address Learning Goes Wrong: Inducing Forwarding Loops and DoS Amplification in SDN." This appendix provides a walkthrough of the artifact, including the installation, execution, and expected results for each experiment. Note that for reviewers' convenience, we provide a **prebuilt VM image** that contains all dependencies and the artifact itself, so that the reviewers can run the experiments without manual installation. This artifact's main function is to instantiate the LoopGen attack and reproduce its (1) feasibility, (2) amplification effect, (3) cost, (4) stealthiness, and (5) defense evaluations.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact intentionally sends spoofed packets, induces forwarding loops, and creates congestion inside emulated SDN environments. Several workflows require `sudo` privileges. The artifact does not require human subjects and does not collect personal or sensitive user data.

A.2.2 How to access

The artifact is archived at https://doi.org/10.5281/zenodo.20307092. The link of our VM image is also in its `./README.md`.

A.2.3 Hardware dependencies

Please use Windows system to run the VM. We validated the artifact on a standard x86_64 Linux VM with at least 8 CPU cores, 16 GB RAM, and roughly 40 GB of free disk space. The VM runs on VirtualBox. We think a powerful laptop or PC is necessary to run the VirtualBox VM smoothly. Note that we also used Barefoot Tofino switch in a minor case study, but it is not required for the main evaluation workflow. We are applying for external access to the Tofino switch, and we will provide the details for reviewers if the access is granted.

A.2.4 Software dependencies

In the released VM image, all dependencies are pre-installed.

A.2.5 Benchmarks

We only use one public traffic dataset `202201011400.pcap` from the MAWI Working Group Traffic Archive (https://mawi.wide.ad.jp/mawi/samplepoint-F/2022/202201011400.html)

A.3 Set-up

A.3.1 Installation

1. Download and unpack the artifact.
2. Open the `Readme.md` in the root directory, install VirtualBox and download our VM image.
3. Use VirtualBox to boot the VM image, which already contains all dependencies and the artifact itself.
4. Choose the claims in Figure 1 and read the corresponding directory in the artifact. The `readme.md` in each directory contains the instructions[1].

A.3.2 Basic Test

Please see **E1.1** in Section A.4.2.

[1]We highly recommend readers to read the `Readme.md` files in the artifact, because they contain screenshots for step-by-step instructions while this appendix do not contain them due to space limitation.

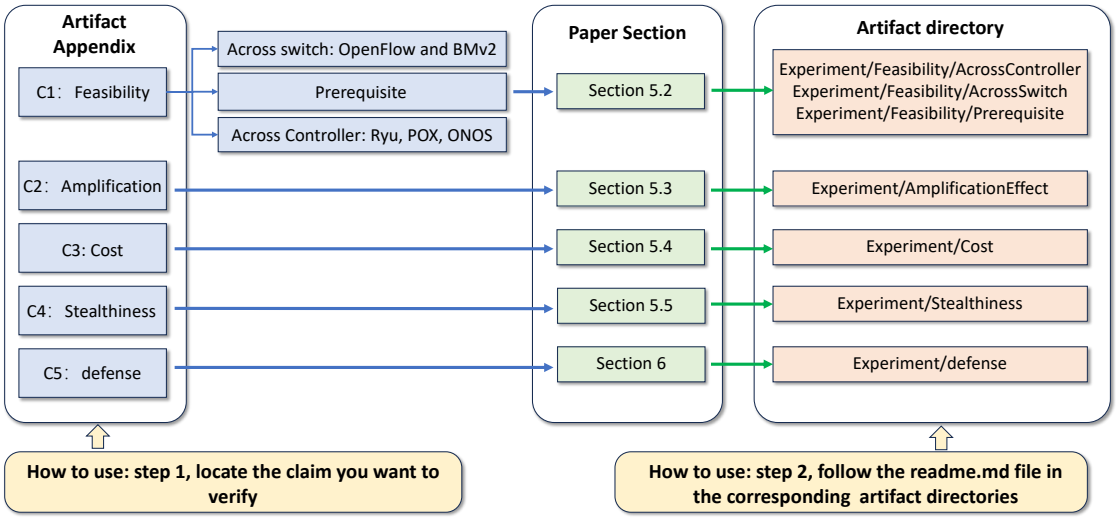


Figure 1: Organization of the artifact and its relation to the paper.

A.4 Evaluation workflow

A.4.1 Major Claims

Our major claims are summarized in Figure 1 and the following list. Each claim is verified by one or more experiments in the artifact.

(C1) *Feasibility: (1) its prerequisite in real-world topologies, (2) its feasibility on different controllers (ryu, pox, and onos), and (3) its feasibility on two types of switches (OpenFlow and programmable switch, i.e., BMv2).*

(C2) *Amplification Effect: LoopGen can significantly amplify traffic and degrade network performance (RTT and Bandwidth).*

(C3) *Low Cost: LoopGen achieves comparable or stronger impact than a non-looping DoS attack while requiring a much lower sending rate.*

(C4) *Stealthiness: LoopGen are not detected by existing detection mechanisms.*

(C5) *Our two defenses can detect LoopGen.*

A.4.2 Experiments

(E1.1) Prerequisite feasibility [5 human-minutes + 10 compute-minutes]. Verify Claim (C1). We recommend to read the `./Experiment/Feasibility/Prerequisite/readme.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. Open a terminal (entering the `p4` directory by default) and run

```
cd Prerequisite
```

For the probability that hosts form a loop structure, run

```
python3 Phost.py
```

For the probability in Leaf-Spine and Fat-Tree, run

```
python3 LSandFT.py
```

Results. The scripts print the probability of loop-feasible host placements in the evaluated topologies. These results are identical to the `./Experiment/Feasibility/Prerequisite/RawData&Figures/loopprobability.ipynb`. The `.ipynb` file can be opened in Visual Studio Code.

(E1.2) Feasibility on Ryu [5 human-minutes + 5 compute-minutes]. Verify Claim (C1). We recommend to read the `./Experiment/Feasibility/AcrossController/Ryu/readme.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default) 2. Start the controller

```
ryu-manager ryu/ryu/app/simple_switch_13.py
```

3. Open a new terminal and start mininet topology

```
cd loopgenexp/ControllerFeasibility-ryu/
sudo python3 -E LoopGen_topo.py
```

4. Open the host terminals in mininet

```
mininet> xterm h1 h2 h3
```

5. Send packets in each host terminal. Note: Don't hit Enter right away. Type the command on all three first, then press Enter on all of them at once.

```
h1> python3 hack_h1_auto.py
h2> python3 hack_h2_auto.py
h3> python3 hack_h3_auto.py
```

Please **wait** until all hosts finshed the attack.

6. Open a new terminal and start wireshark. Then, monitor the **`s1-eth2`** port.

```
sudo wireshark
```

Results. Monitor `s1-eth2` with Wireshark. The expected outcome is a persistent stream of looping ARP packets.

(E1.3) Feasibility on POX [5 human-minutes + 5 compute-minutes]. Verify Claim (C1). We recommend to read the `./Experiment/Feasibility/AcrossController/POX/l2/README.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default) 2. Start the controller

```
cd pox
./pox.py log.level -DEBUG openflow.of_01 -port=6633
forwarding.l2_learning
```

3. Open a new terminal and start Mininet topology

```
cd loopgenexp/ControllerFeasibility-pox/l2/
sudo python3 -E LoopGen_topo.py
```

4. Open the host terminals in Mininet

```
mininet> xterm h1 h2 h3
```

5. Send packets in each host terminal. Note: Don't hit Enter right away. Type the command on all three first, then press Enter on all of them at once.
In h1 xterm, run

```
python3 hack_h1_auto.py
```

In h2 xterm, run

```
python3 hack_h2_auto.py
```

In h3 xterm, run

```
python3 hack_h3_auto.py
```

Please **wait** until all hosts finish the attack.
6. Open a new terminal and start Wireshark. Then, monitor the **`s1-eth2`** port.

```
sudo wireshark
```

7. For convenience, filter only ARP packets. You can see that the attack packets are looped continuously. **Results.** Monitor `s1-eth2` with Wireshark. The expected outcome is a persistent stream of looping ARP packets.

(E1.4) Feasibility on ONOS [10 human-minutes + 10 compute-minutes]. Verify Claim (C1). We recommend to read the `./Experiment/Feasibility/AcrossController/ONOS/README.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default) 2. Enter the `onos` directory and start ONOS

```
cd onos
bazel run onos-local
```

3. Open a new terminal, enter the `onos` directory, and connect to the ONOS CLI

```
cd onos
./tools/test/bin/onos localhost
```

4. Activate the required ONOS apps

```
app activate org.onosproject.openflow
app activate org.onosproject.drivers
app activate org.onosproject.fwd
wipe-out please; app deactivate
org.onosproject.fwd; app activate
org.onosproject.fwd
```

5. Open a new terminal, start the Mininet topology, and open the host terminals

```
cd loopgenexp/ControllerFeasibility-onos/
sudo python3 -E LoopGen_topo.py
xterm h1 h2 h3
```

6. Launch the attack in each host terminal. Note: Don't hit Enter right away. Type the command on all three first, then press Enter on all of them at once.
In h1 xterm, run

```
python3 attack_l2_timed.py h1-eth0
00:00:00:00:00:02
```

In h2 xterm, run

```
python3 attack_l2_timed.py h2-eth0
00:00:00:00:00:03
```

In h3 xterm, run

```
python3 attack_l2_timed.py h3-eth0
00:00:00:00:00:01
```

7. Then, in the Mininet terminal:

```
mininet> h1 python3 -c "from scapy.all import sendp, Ether,
IP, UDP;
sendp(Ether(src='00:00:00:00:00:01',
dst='00:00:00:00:00:08') /
IP(dst='10.0.0.8') / UDP(dport=9999), iface='h1-eth0',
count=1)"
```

8. Open a new terminal and start Wireshark

```
sudo wireshark
```

Results. Monitor the corresponding switch interface with Wireshark. The expected outcome is that the injected packet destined to `10.0.0.8` keeps looping after the trigger packet is sent.

(E1.5) BMv2 feasibility [5 human-minutes + 5 compute-minutes]. Verify Claim (C1). We recommend to read the `./Experiment/Feasibility/AcrossSwitch/readme.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default) 2. Enter the experiment directory and start the topology

```
cd tutorials/exercises/LoopGen/
make
```

If any compiling errors occur, try

```
make clean
make
```

3. Open another terminal and start the controller

```
cd tutorials/exercises/LoopGen/
sudo python3 controller.py
```

4. In the Mininet terminal, open the host terminals. Note: Don't hit Enter right away. Type the command on all three first, then press Enter on all of them at once.

```
xterm h1 h2 h3
```

In h1 xterm, run

```
python3 hack_h1_auto.py
```

In h2 xterm, run

```
python3 hack_h2_auto.py
```

In h3 xterm, run

```
python3 hack_h3_auto.py
```

Please **wait** until they finish the attack.
5. Open a new terminal and start Wireshark. Then, monitor the **`s1-eth2`** port.

```
sudo wireshark
```

Results. Monitor `s1-eth2` with Wireshark. The expected outcome is again a continuous loop of attack packets on the monitored interface. This confirms that the same attack logic works on the programmable software-switch setting used by the rest of the BMv2-based experiments. The sibling `AcrossSwitch-Tofino/` note documents an additional hardware case study, but it is not the base of any other experiments; the same forwarding behavior is already mirrored by this BMv2 experiment.

(E2.1) Evaluate amplification factor [10 human-minutes + 10 compute-minutes + up to 40 GB disk for packet captures]. Verify Claim (C2). We recommend to read the `./Experiment/Feasibility/AmplificationEffect/readme.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default). 2. Enter the experiment directory and start the topology.

```
cd tutorials/exercises/AmplificationEffect/
make
```

3. Open Wireshark **before** sending any packets. Then, monitor the **`s1-eth2`** port.

```
sudo wireshark
```

You can see that there is no packet at this point. In the Mininet terminal, open `h1` and send the test packet.

```
mininet> xterm h1
h1> python3 send.py
```

`send.py` sends **one** packet whose TTL is `255`.

Results. Then, you can see that `85` packets are captured by Wireshark, which means that this packet was looped `85` times.

(E2.2) Evaluate RTT [10 human-minutes + 10 compute-minutes + up to 40 GB disk for packet captures]. Verify Claim (C2). We recommend to read the `./Experiment/Feasibility/AmplificationEffect/readme.md` in the artifact.

1. First, open an `h1` xterm and use `tcpreplay` to inject attack packets.

```
mininet> xterm h1
h1> tcpreplay -i h1-eth0 -p 300 -l 0 LoopGen.pcap
```

2. Then, open a new `h1` xterm and ping `h2`.

```
h1> ping 10.0.0.2
```

Results. You can see RTT increases significantly.

(E2.3) Evaluate throughput [10 human-minutes + 10 compute-minutes + up to 40 GB disk for packet captures]. Verify Claim (C2). We recommend to read the `./Experiment/Feasibility/AmplificationEffect/readme.md` in the artifact.

1. First, open an `h1` xterm and use `tcpreplay` to inject attack packets.

```
mininet> xterm h1
h1> tcpreplay -i h1-eth0 -p 300 -l 0 LoopGen.pcap
```

2. Then, open a new `h1` xterm and an `h2` xterm.

```
mininet> xterm h1 h2
```

3. In the `h2` xterm, start an `iperf` TCP server.

```
h2> iperf -s
```

4. In the new `h1` xterm (do **not** change the other `h1` xterm running `tcpreplay`), run

```
h1> iperf -c 10.0.0.2
```

5. Then, on the `h2` xterm, you can see the evaluated bandwidth (the maximum is `10` Mbps, so we can calculate the degradation).

Results. Throughput is `8.33` Mbps corresponds to a `14.7`% degradation from the `9.76` Mbps normal link rate.

If you want to change topologies, edit the `Makefile`. The default topology is `network-BAP.py`. The other topology scripts are included in the same directory.

(E3) Cost comparison against normal DoS [20 human-minutes + 10 compute-minutes]. Verify Claim (C3). We also recommend reviewers to read the `./Experiment/Cost/readme.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default). 2. Enter the experiment directory and start the topology.

```
cd tutorials/exercises/Cost/
make
```

• **Launch LoopGen.**

1. First, open an `h1` xterm and use `tcpreplay` to inject attack packets.

```
mininet> xterm h1
h1> tcpreplay -i h1-eth0 -p 300 -l 0 LoopGen.pcap
```

2. Then, open a new h1 xterm and ping h2.

```
h1> ping 10.0.0.2
```

• **Launch normal DoS.**

1. First, open an h1 xterm and use tcpreplay to inject attack packets. Use normalDoS.pcap instead of LoopGen.pcap, and increase the replay speed gradually (for example, 5000, 10000, 15000, 20000, ...) until the latency reaches the same level as under LoopGen.

```
mininet> xterm h1
h1> tcpreplay -i h1-eth0 -p 5000 -l 0
normalDoS.pcap
```

2. Then, open a new h1 xterm and ping h2.

```
h1> ping 10.0.0.2
```

Results. Compare the latency under LoopGen with the latency under normal DoS while increasing the replay rate. The expected outcome is that the non-looping DoS requires a much higher replay rate than LoopGen to reach a comparable impact, demonstrating LoopGen's cost-effectiveness.

(E4.1) Stealthiness against Identity-verification [5 human-minutes + 5 compute-minutes]. Verify Claim (C4). Read ./Experiment/Stealthiness/IdentityVerification/readme.md for detailed instructions.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the p4 directory by default). 2. Start the identity-verification application.

```
ryu-manager ryu/ryu/app/switchfix.py
```

3. Open a second terminal and start the Mininet topology.

```
cd loopgenexp/identityverification/
sudo python3 topology.py
```

4. Open two sets of host terminals and start the responder scripts in one set.

```
mininet> xterm h1 h2 h3
mininet> xterm h1 h2 h3
```

In the first h1 xterm, run

```
sudo python3 responder1.py h1-eth0
```

In the first h2 xterm, run

```
sudo python3 responder2.py h2-eth0
```

In the first h3 xterm, run

```
sudo python3 responder3.py h3-eth0
```

5. In the second set of host terminals, start the LoopGen attack scripts. Note: Don't hit Enter right away. Type the command on all three first, then press Enter on all of them at once.

In the second h1 xterm, run

```
sudo python3 hack_h1_auto.py
```

In the second h2 xterm, run

```
sudo python3 hack_h2_auto.py
```

In the second h3 xterm, run

```
sudo python3 hack_h3_auto.py
```

6. Open Wireshark and monitor s1-eth2.

Results. Monitor s1-eth2 with Wireshark. The expected outcome is that packets still loop continuously, indicating that this identity-verification defense does not stop LoopGen in the tested setting.

(E4.2) Stealthiness against EventScope [10 human-minutes + 10 compute-minutes]. Verify Claim (C4). Read ./Experiment/Stealthiness/EventScope/README.md for detailed instructions.

Preparation. Please see Section A.3.1. The default JDK in the image is JDK 11, so first switch to JDK 8.

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
java -version
```

Execution. Run the static analysis.

```
cd ~/EventScope-master
java -jar $ANALYZER_JAR
```

Open the generated event-flow-graph.pdf report.

Results. The expected outcome is that EventScope completes and generates the PDF analysis report for org.onosproject.fwd.ReactiveForwarding. It can be seen that the attack did not introduce new event listeners or data plane interaction paths in the graph. This is because it merely added static mappings to the existing event handling logic, thereby altering the forwarding semantics but not the event flow structure.

(E4.3) Stealthiness against SVHunter [5 human-minutes + 5 compute-minutes]. Verify Claim (C4). Read ./Experiment/Stealthiness/SVHunter/README.md for detailed instructions.

Preparation. Please see Section A.3.1. The default JDK in the image is JDK 11, so first switch to JDK 8.

```
export JAVA_HOME=/usr/lib/jvm/java-8-openjdk-amd64
export PATH=$JAVA_HOME/bin:$PATH
java -version
```

Execution. 1. Compile the analysis tool.

```
cd ~/SVHunter-master/tracer
mvn clean package
```

2. Run the analysis on the ONOS source tree.

```
java -jar target/SVHunter-Tracer-0.1-SNAPSHOT-jar-with-dep
endencies.jar \
~/onos \
config/onosEnv.cfg \
config/onosSensitiveMethodList.xml \
config/onosDFMethodList.xml
```

3. Inspect the result file under ~/SVHunter-master/tracer/backwardFlow.

Results. The expected outcome is that SVHunter completes successfully and the result file does not report an alert for `org.onosproject.fwd.ReactiveForwarding`, indicating that this workflow does not detect LoopGen through SVHunter.

(E4.4) Stealthiness against Lemon [5 human-minutes + 5 compute-minutes]. Verify Claim (C4). Read `./Experiment/Stealthiness/Lemon/readme.md` for detailed instructions.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default). 2. Start the BMv2-based Mininet experimental environment.

```
cd tutorials/exercises/lemon/
sudo p4run
```

3. Start the attack. The file `output-1000B.pcap` is the attack traffic set, and `202201011400.pcap` is the background traffic.

```
mininet> xterm h1 h1
h1> tcpreplay -i h1-eth0 --duration=4
output-1000B.pcap
h1> tcpreplay -i h1-eth0 --pps=3200 --duration=4
--loop=0 202201011400.pcap
```

4. Open a new terminal and start the Lemon controller.

```
sudo python3 controller.py
```

Inspect the controller output. The destination `10.0.0.8` does not appear in the output.

5. Then open a new terminal and start Wireshark.

```
sudo wireshark
```

Verify the loop by sending only one attack packet.

```
h1> tcpreplay -i h1-eth0 -L 1 output-1000B.pcap
```

Results. The expected outcome is that Lemon does not report the LoopGen destination in the controller output, while Wireshark still shows that the attack packet is looped multiple times. This indicates that the attack remains stealthy to Lemon in this workflow.

(E4.5) Stealthiness against SISTAR [5 human-minutes + 5 compute-minutes]. Verify Claim (C4). Read `./Experiment/Stealthiness/SISTAR/readme.md` for detailed instructions.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default). 2. Start the BMv2-based Mininet experimental environment.

```
cd tutorials/exercises/sistar/
make
```

3. Open a new terminal and install the flow rules for the `S1-S2-S3` triangle loop topology.

```
cd tutorials/exercises/sistar/
simple_switch_CLI --thrift-port 9090 <<< "table_add
MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.99.99/32 =>
00:00:00:00:02:00 2"
simple_switch_CLI --thrift-port 9091 <<< "table_add
MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.99.99/32 =>
00:00:00:00:03:00 3"
simple_switch_CLI --thrift-port 9092 <<< "table_add
MyIngress.ipv4_lpm MyIngress.ipv4_forward 10.0.99.99/32 =>
00:00:00:00:01:00 4"
```

4. Deploy the defense mechanism.

```
bash run_defense.sh
```

5. Open two xterms and send the background traffic and the attack traffic.

```
tcpreplay -i eth0 -p 15000 -L 10000
./202201011400.pcap &
tcpreplay -i eth0 -p 1000 -l 20000 attack.pcap
```

Then open Wireshark and monitor `s1-eth2`.

Results. The expected outcome is that, even after the SISTAR defense is deployed, Wireshark still shows looped attack packets whose destination is `10.0.99.99`. This indicates that the attack remains stealthy to SISTAR in this workflow.

(E5) Controller-side countermeasures [10 human-minutes + 10 compute-minutes]. Verify the second half of Claim (C5). We recommend to read the `./Experiment/defense/readme.md` in the artifact.

Preparation. Please see Section A.3.1.

Execution. 1. Open a terminal (entering the `p4` directory by default).

• **Loop-Detection Countermeasure.**

1. Start the Ryu controller with topology observation enabled.

```
ryu-manager --observe-links
ryu/ryu/app/loop_detect_13.py
```

2. Open a new terminal and start the Mininet topology.

```
cd loopgenexp/ControllerFeasibility-ryu/
sudo python3 -E LoopGen_topo.py
```

3. Open the host terminals in Mininet.

```
mininet> xterm h1 h2 h3
```

4. Send packets in each host terminal.

In `h1`, run

```
python3 hack_h1_auto.py
```

In `h2`, run

```
python3 hack_h2_auto.py
```

In `h3`, run

```
python3 hack_h3_auto.py
```

5. Inspect the controller output.

• **Time-Constraint Countermeasure.**

1. Start the Ryu controller with the time-constraint defense and topology observation enabled.

```
ryu-manager --observe-links
ryu/ryu/app/time_constraint_13.py
```

2. Open a new terminal and start the Mininet topology.

```
cd loopgenexp/ControllerFeasibility-ryu/
sudo python3 -E LoopGen_topo.py
```

3. Open the host terminals in Mininet.

```
mininet> xterm h1 h2 h3
```

4. Send packets in each host terminal.
In `h1`, run

```
python3 hack_h1_auto.py
```

In `h2`, run

```
python3 hack_h2_auto.py
```

In `h3`, run

```
python3 hack_h3_auto.py
```

5. Inspect the controller output.
Results. The expected outcome is that both controller-side countermeasures detect or suppress the LoopGen attack rather than allowing an endless loop to persist. The corresponding controller outputs should indicate successful detection or suppression for the two methods.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: DNS Cache Poisoning Like it's 2006

Omer Ben-Simhon
Hebrew University of Jerusalem

Amit Klein
Hebrew University of Jerusalem

A Artifact Appendix

A.1 Abstract

This artifact accompanies the paper *"DNS Cache Poisoning Like it's 2006"* and demonstrates practical DNS cache poisoning attacks against vulnerable versions of the BIND 9 recursive resolver. The artifact implements two attack variants—*RRset* and *RRset-ANY*—and explains how to build a complete, self-contained laboratory environment for reproducing the paper's core claims. Using a controlled four-host DNS setup, the artifact allows evaluators to reproduce successful cache poisoning under realistic resolver configurations. The testing instructions in the artifact are designed to reproduce the results described in Section 5.2 of the paper ("Baseline experiments"), and specifically the RRset and RRset-ANY tests against BIND 9.20 (see Table 2 in the paper). Other experiments are not explicitly included, though a knowledgeable tester can probably adapt the setup to test additional scenarios with some effort.

A.2 Description & Requirements

This artifact demonstrates practical DNS cache poisoning attacks against vulnerable BIND 9 recursive resolvers in a controlled laboratory environment. The attack is orchestrated through the interaction of four distinct hosts, each playing a well-defined role in the DNS resolution process and the attack workflow. The artifact does not recreate the exact Azure cloud environment used in the paper's large-scale experiments. Instead, it reproduces the same attack logic and outcomes in a controlled local (LAN or virtualized) setup. In Azure, cloud anti-spoofing restrictions required us to use the "spoofing-at-the-destination" technique as an environmental workaround. This does not change the attack itself, but only how spoofed packets are injected under cloud constraints. The artifact recommends a local setup because it avoids these cloud-specific restrictions and simplifies reproduction, while preserving the core attack behavior, resolver interactions, and success conditions described in the paper.

System architecture. The experimental setup consists of the following four machines (we denote each machine with a bracketed, gothic, colored letter here, and later use this notation when prescribing shell commands, to clarify in which machine(s) they should run):

- **Resolver (BIND)**: a BIND 9 recursive resolver that serves DNS queries from clients and maintains a cache. This resolver is the target of the cache poisoning attack. Denoted by $\langle\mathcal{R}\rangle$.
- **Legitimate Authoritative Name Server (ANS)**: an authoritative DNS server for the domain `victim.example`, which serves legitimate DNS records – this is the domain the attacker targets. Denoted by $\langle\mathcal{V}\rangle$.
- **Attacker Authoritative Name Server (ANS)**: an authoritative DNS server for the domain `attacker.example`, controlled by the attacker and used to provide malicious resource records during the attack. Denoted by $\langle\mathcal{A}\rangle$.
- **Attacker Client**: an attacker client machine that triggers DNS resolution at the resolver and launches the cache poisoning attack by sending legitimate queries alongside spoofed DNS responses. Denoted by $\langle\mathcal{C}\rangle$.

All experiments are conducted in an isolated network under the evaluator's control.

Poisoning target. The concrete poisoning target in the artifact is the DNS name:

`www0.victim.example`

Prior to the attack, this name (more accurately, its DNS A record) is legitimately resolved by the resolver to an IPv4 address served by the legitimate authoritative server:

`www0.victim.example` → `198.51.100.10`

The goal of the attack is to poison the resolver's cache such that subsequent queries for this name are resolved to an attacker-chosen IPv4 address:

`www0.victim.example` → `198.51.100.66`

The artifact demonstrates this by first seeding the resolver cache with the genuine record and then racing spoofed responses against the legitimate response from the legitimate

authoritative server, as described in the paper. Successful poisoning is verified by querying the resolver after the attack, and observing that it returns the attacker-controlled address from its cache.

There are two hard-wired poisoned answers involved. The first one (with TTL=35s) is the "Kaminsky answer" (sent by the attacker with the predicted TXID and UDP port) which contains a malicious NS record (+glue) for `www0`, pointing at the attacker client IP address. The second poisoned answer (with TTL=30s) is the "payload answer" generated for the `A?www0` query sent by the BIND9 resolver to the attacker, once the A record for `www0` expires from BIND9's cache. The poisoned IPv4 address (`198.51.100.66`) used in this answer is hard-coded in the script for reproducibility. In real attacks, attackers would typically use a much larger TTL value in the payload answer, to maximize persistence.

A.2.1 Security, privacy, and ethical concerns

This artifact sends *spoofed UDP DNS packets* and deliberately races legitimate DNS responses. Running the artifact on production networks or public infrastructure may violate acceptable-use policies and local regulations and laws. Evaluators **must** execute the artifact only on isolated laboratory networks (physical or virtual) they fully own and fully under their control. The artifact does not exfiltrate data, access third-party systems, or perform persistence beyond temporary DNS cache entries.

A.2.2 How to access

The artifact is archived on Zenodo. Once the paper is published, the artifact will be accessible via the following stable reference:

`https://doi.org/10.5281/zenodo.17762025`

A.2.3 Hardware dependencies

The artifact runs on commodity x86-64 hardware. Evaluators may use either physical machines or virtual machines.

The attack requires **spoofed UDP packets** to reach the resolver. Many ISPs and cloud providers (e.g., public IaaS platforms such as Azure) block spoofed packets, even on internal virtual networks. **Strong recommendation:** perform all testing on a physical LAN, or a locally virtualized LAN (e.g., VMware, VirtualBox, or QEMU).

In addition, don't use networks that have NATs or any other functionality that may alter packets. For example, don't use VMware's NAT mode; you should use e.g. bridged mode instead.

For consistency with the experimental results reported in the paper, we strongly recommend using **Ubuntu 24.04** on all machines and using **BIND 9** as the authoritative name server software on the legitimate authoritative server and the attacker authoritative server machines.

BIND versions. Only vulnerable BIND 9 versions can reproduce the attacks.
Vulnerable (usable) versions: BIND 9.18.0–9.18.39, BIND 9.20.0–9.20.13, BIND 9.21.0–9.21.12.
Patched versions (not vulnerable; do not use): BIND 9.18.41+, BIND 9.20.15+, BIND 9.21.14+.
The recommended version for evaluation is **BIND 9.20.13**. As far as we know, from the perspective of the vulnerabilities described in the paper, BIND 9.20.13 behaves identically to BIND 9.20.11 (which is the version paper baseline experiments use). Instructions for downloading and building this BIND version are provided in subsubsection A.3.1.

System packages. The following packages are required on all machines: **build-essential, gcc, g++-10, make, python3 (3.10+), python3-pip, python3-venv, python3-numpy, python3-dns, python3-dnspython, python3-scapy, libssl-dev, wget, curl, git, net-tools, dnsutils, tcptraceroute**.

Additional build dependencies are required on machines running BIND 9 (required for all of the machines except the attacker client): **libuv1-dev, libnghttp2-dev, libcap-dev, libjson-c-dev, pkg-config, autoconf, automake, libtool, liburcu-dev**.

A.2.4 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

After installing the required packages for each machine as described in previous section, proceed to the instruction in this section.

Installing BIND 9.20.13 from source. ⟨$\mathcal{A}$⟩ ⟨$\mathcal{V}$⟩ ⟨$\mathcal{R}$⟩
BIND 9.20.13 can be obtained from the ISC archive: `https://ftp.isc.org/isc/bind9/9.20.13/bind-9.20.13.tar.xz`
Assuming BIND 9 was downloaded to to a file named `bind-9.20.13.tar.xz`, unpack, build, and install it:

```
tar -xf bind-9.20.13.tar.xz
cd bind-9.20.13
./configure
make
sudo make install
sudo ldconfig -v
sudo chmod 777 .
```

System configuration. ⟨$\mathcal{A}$⟩ ⟨$\mathcal{V}$⟩ ⟨$\mathcal{R}$⟩ The built-in Linux stub resolver must be disabled on all machines:

```
sudo systemctl stop systemd-resolved
sudo systemctl disable systemd-resolved
```

⟨$\mathcal{R}$⟩ Between experiments, the resolver must be restarted to clear cached state. If running `named` from source (as suggested above) run:

```
sudo killall named
```

Configuration files and IPv4 address placeholders. All configuration files provided in the artifact contain placeholder IP addresses that **must be replaced** with their corresponding concrete IPv4 addresses before running the experiments: `<resolver_ip>`, `<legitimate_ans_ip>`, `<attacker_ans_ip>`, `<attacker_client_ip>`.

Machine-specific configuration.

Configuration template (DNS servers). For each DNS server (authoritative or resolver), the setup follows the same procedure: (i) copy the relevant configuration files to `/etc`, (ii) replace all placeholder IP addresses, and (iii) start `named` with the corresponding main configuration file.

Attacker Authoritative Name Server (ANS): ⟨$\mathcal{A}$⟩ Configuration files: `attacker.named.conf`, `attacker.named.conf.local`, `attacker.example.zone`. Note that the attacker ANS IP address appears twice and must be updated consistently. Start the server using:

```
sudo named -4 -c /etc/attacker.named.conf -g
```

Legitimate Authoritative Name Server (ANS): ⟨$\mathcal{V}$⟩ Configuration files: `legitimate.named.conf`, `legitimate.named.conf.options`, `legitimate.named.conf.local`, `victim.example.zone`. Start the server using:

```
sudo named -4 -c /etc/legitimate.named.conf -g
```

Resolver (BIND): ⟨$\mathcal{R}$⟩ Configuration files: `resolver.named.conf`, `resolver.named.conf.options`, `resolver.named.conf.local`, `resolver.example.zone`. Start the resolver using:

```
sudo named -4 -c /etc/resolver.named.conf -g
```

Attacker Client: ⟨$\mathcal{C}$⟩ All artifact files should be copied to this machine. Build the attacker's script components using:

```
make all
```

⟨$\mathcal{C}$⟩ Run the attack script with the appropriate arguments, e.g.:

```
sudo python3 attacker_client.py ...
```

(Concrete command lines are given in subsubsection A.4.2)

Network timing configuration. To create a reproducible attack window on low-latency networks, evaluators are recommended to delay responses from the legitimate authoritative server: ⟨$\mathcal{V}$⟩

```
sudo tc qdisc add dev <interface_name> root netem
↪ delay 80ms
```

Where `<interface_name>` is the interface used to send traffic to the other hosts.

After completing the experiments, the delay must be removed: ⟨$\mathcal{V}$⟩

```
sudo tc qdisc del dev <interface_name> root netem
```

A.3.2 Basic Test

⟨$\mathcal{R}$⟩ Before running the attack, evaluators should verify that the resolver uses the expected ephemeral port range:

```
sysctl net.ipv4.ip_local_port_range
```

which should return `32768 60999`.

⟨$\mathcal{C}$⟩ To verify correct installation, evaluators should query the resolver before launching the attack:

```
dig @<resolver_ip> A www0.victim.example
```

Expected output (before attack):

```
www0.victim.example.  <TTL>  IN  A  198.51.100.10
```

This confirms that the resolver correctly fetches and caches the legitimate record.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): *A remote attacker can poison the cache of a vulnerable BIND 9 recursive resolver using an RRset-based attack over UDP or TCP.* This claim is demonstrated by experiment (E1) and corresponds to the RRset attack described in Section 4.2.1 of the paper and evaluated in Section 5.2.

(C2): *A remote attacker can poison the cache of a vulnerable BIND 9 recursive resolver using the RRset-ANY attack variant over both UDP and TCP.* This claim is demonstrated by experiment (E2) and corresponds to the RRset-ANY attack described in Section 4.2.1 of the paper and evaluated in Section 5.2.

A.4.2 Experiments

RRset (E1) / RRset-ANY (E2) Attacks, over TCP / UDP [1 human-minute per each combination]

Herein we describe four experiments (combinations): Two E1 combinations (RRset over TCP and RRset over UDP), and two E2 combinations (RRset-ANY over TCP and RRset-ANY over UDP).

Preparation: Ensure that all four machines are running and configured as described above in subsubsection A.3.1.

Execution: ⟨C⟩ On the attacker client machine, from within the artifact directory, run the following command template, setting the parameters as follows:

- **`-mode`** to `rrset` or `rrset_any`
- **`-protocol`** to `tcp` or `udp`
- **`-rrset_domain`** to `x.attacker.example` (RRset), or to `x1.attacker.example` / `x2.attacker.example` for RRset-ANY over TCP / UDP, respectively

```
sudo python3 attacker_client.py \
  --mode <rrset|rrset_any> \
  --protocol <tcp|udp> \
  --iterations 1 \
  --resolver_ip <resolver_ip> \
  --attacker_ans_ip <attacker_ans_ip> \
  --attacker_ans_domain attacker.example \
  --victim_ans_domain victim.example \
  --legitimate_ans_ip <legitimate_ans_ip> \
  --logging console \
  --rrset_domain <x|x1|x2>.attacker.example \
  --public_ip <attacker_client_ip>
```

The script automatically seeds the resolver cache with the legitimate record, executes the attack, and terminates after approximately 35 seconds.

Results: During execution, the script prints the message:

```
Attack Succeeded!
```

⟨C⟩ Immediately after the script finishes, query the resolver:

```
dig @<resolver_ip> -4 A www0.victim.example
```

The expected response is a poisoned cache entry:

```
www0.victim.example.   <some_TTL>   IN   A
↪  198.51.100.66
```

A.5 Notes on Reusability

This artifact can be reused to evaluate:

- Different BIND 9 versions and configurations
- TCP vs. UDP transport effects
- Defensive mechanisms (e.g., patched BIND versions)

The modular configuration files allow easy adaptation to additional experimental scenarios.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Side-Channel Attacks on Open vSwitch

Daewoo Kim
University of Waterloo

Sihang Liu
University of Waterloo

A Artifact Appendix

A.1 Abstract

Virtualization is widely adopted in cloud systems to manage resource sharing among users. A virtualized environment usually deploys a virtual switch within the host system to enable virtual machines to communicate with each other and with the physical network. The Open vSwitch (OVS) is one of the most popular software-based virtual switches. It maintains a cache hierarchy to accelerate packet forwarding from the host to virtual machines.

We provide code that characterizes the OVS caching system from a security perspective and identifies key attack primitives. Based on these primitives, the artifact demonstrates three remote attacks that break isolation guarantees in virtualized environments. First, the artifact enables a remote covert channel in which a sender transmits bits from outside the server while a receiver observes cache behavior inside the server. Second, it presents a novel header recovery attack that probes microflow cache entries and recovers a remote user's IP address and port number using the known hash function. Third, the artifact demonstrates a remote packet-rate monitoring attack that exploits megaflow subtables to infer a remote victim's packet rate. Finally, three mitigation strategies that successfully prevent attacks are included.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

All experiments are done in local, isolated environments. We do not identify any risk to users of the artifact associated with this artifact evaluation procedure.

A.2.2 How to access

Our artifact can be downloaded from Zenodo at https://doi.org/10.5281/zenodo.17965902.

A.2.3 Hardware dependencies

The experiments require the following hardware platform:

- Two servers, each equipped with at least a 1 Gbps NIC.
- A 1 Gbps one-hop ethernet connection between the two servers.

A.2.4 Software dependencies

The experiments require the following software platform:

- Ubuntu 22.04.
- Linux kernel version 6.5.0.
- Open vSwitch version 2.17.9 (`https://www.openvswitch.org/download/`).
- DPDK version 21.11.9 (`https://core.dpdk.org/download/`).
- Memcached version 1.6.21 (`https://github.com/memcached/memcached/wiki/ReleaseNotes`).

A.2.5 Benchmarks

The experiments require the following workloads:

- **ClassBench-ng** (`https://classbench-ng.github.io/`).
- **UNSW-NB15** (`https://research.unsw.edu.au/projects/unsw-nb15-dataset`).

A.3 Set-up

A.3.1 Installation

`Sudo` permission may be required to execute the following commands. First, run `install_ovs_dpdk.sh` on server1 to install Open vSwitch version 2.17.9 with DPDK version 21.11.9. Then, run `vm_setup.sh` on server1 to create two virtual machines and attach them to an OVS bridge.

A.3.2 Basic Test

We provide a simple test to verify that all required dependencies are installed correctly. Run `Basic_Test/0_Basic_Test.sh`. The output should include a measured RTT.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Figure 7 presents the RTT distributions for microflow cache hits and megaflow cache hits, demonstrating that their RTTs are distinguishable.
(C2): Table 2 presents the results of covert data transmission using the microflow cache and the megaflow cache.
(C3): Table 4 presents the results of recovering the user's IP address and port number.
(C4): Figure 20 presents the results of monitoring the user's packet rate.
(C5): Table 5 presents the evaluation results of three defense mechanisms.

A.4.2 Experiments

(E1): [Characterization] [1 compute-hour]: Characterization logs the RTT to a remote server by sending Memcached GET requests. It measures RTTs in the case of a microflow cache hit and RTTs in the case of a megaflow cache hit.
Preparation: Run `vm_setup.sh` on server1.
Execution: Run `Characterization/1_Characterization.sh` on server2.
Results: The results should match Figure 7. The average RTTs is used as a threshold in the following experiments.

(E2): [Covert Channel] [1 compute-hour]: Covert Channel consists of a sender and a receiver. It randomly generates an 8-bit payload and transmits a 10-bit sequence with a 10 header. To transmit a bit 1, the sender sends packets to the Memcached server; to transmit a bit 0, it remains idle. The receiver continuously measures RTTs to the Memcached server. Reception starts when a high RTT is detected, which indicates the beginning of the header. For each bit, the receiver averages the RTT of 100 measurements, and logs the recovered bit based on the average.
Preparation: Both server1 and server2 run two virtual machines each. On server1, run the receiver VM and a Memcached VM. On server2, run the sender VM and a Memcached VM. Use `vm_setup.sh` to set up the VMs on both servers. Configure Memcached with the -l 0.0.0.0 option so that it listens on all available network interfaces.
Execution: Run `Attacks/2_Covert_channel_receiver.sh` on server1 and `Attacks/2_Covert_channel_sender.sh` on server2.
Results: The results should match Table 2.

(E3): [Header Recovery] [1 compute-hour]: Header recovery first probes all 8192 microflow cache entries by sending packets to the Memcached server and records the RTTs. The two entries with the highest RTTs correspond to entries used by the victim. To identify the remaining victim entries, the code repeats the probing process using different packet flows and identifies another two high RTT entries. Lastly, it reverse-engineers the victim's IP address and port number using the four detected entries.
Preparation: Both server1 and server2 run two virtual machines each. On server1, run the victim VM and a Memcached VM. On server2, run the attacker VM and a Memcached VM. Use `vm_setup.sh` to set up the VMs on both servers. Configure Memcached with the -l 0.0.0.0 option so that it listens on all available network interfaces.
Execution: Run `Attacks/3_Header_recovery_victim.sh` on server1 and `Attacks/3_Header_recovery.sh` on server2.
Results: The results should match Table 4.

(E4): [Rate Monitoring] [1 compute-hour]: Rate monitoring logs the range of monitored packet rates. It periodically sends packets to the Memcached server to control the ordering of megaflow cache subtables. The setup includes four threshold subtables and one subtable shared with the victim. By measuring RTTs to these five subtables, it infers the range of the victim's packet rate. This range is adaptively doubled or halved based on the result from the previous round.
Preparation: Both server1 and server2 run two virtual machines each. On server1, run the victim VM and a Memcached VM. On server2, run the attacker VM and a Memcached VM. Use `vm_setup.sh` to set up the VMs on both servers. Configure Memcached with the -l 0.0.0.0 option so that it listens on all available network interfaces.
Execution: Run `Attacks/4_Rate_monitoring_victim.sh` on server1 and `Attacks/4_Rate_monitoring.sh` on server2.
Results: The results should match Figure 20.

(E5): [Mitigation] [1 human-hour + 3 compute-hour]:
Preparation: Repeat the installation after replacing `dpif-netdev.c` and `dpif-netdev-private-dfc.h`. The logic for the microflow cache and megaflow cache can be modified in these two files.
Execution: Repeat E1, E2, E3, and E4 on server2.
Results: The results should match Table 5.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Concretely Efficient Blind Signatures Based on VOLE-in-the-Head Proofs and the MAYO Trapdoor

Carsten Baum[1], Marvin Beckmann[1], Ward Beullens[2], Shibam Mukherjee[3,4], Christian Rechberger[3,5]

Technical University of Denmark[1], IBM, Zürich[2], Graz University of Technology[3], Know Center, Graz[4], TACEO, Graz[5]

A Artifact Appendix

A.1 Abstract

This artifact appendix contains implementations of our proposed Blind signatures in the main paper. With this artifact, we support our performance claims and provide a functional open access implementation of our work. Easing compilation effort, we also provide a `bench_all_bs.sh` script generating a table corresponding to Table 3 in our main work. A `Docker` file has also been provided to alleviate the complications of package dependencies. The implementation uses optimized C/C++ codebase and combines into a Rust framework. The MAYO implementation is from the most recent NIST submission[1]. The framework for the VOLEitH proofs is from the most recent NIST submission of FAEST[2]. Our ZK-proof is implemented in C++ by modifying the codebase of FAEST. The C/C++ components are made accessible in Rust using foreign function interfaces (FFIs) using the bindgen[3] crate. The high-level aspects of our Blind signature are in Rust. The Rust build system (`cargo`) compiles and wraps the underlying C/C++ code. The code can be run directly through the Rust environment. The protocol logic in Rust can be modified without recompiling the underlying C/C++ code. Any changes to the underlying C/C++ code require a clean build of the FFIs and the Blind signature, as is also done inside the `bench_all_bs.sh` script.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Integrating C/C++ primitives into a Rust environment via FFIs introduces potential memory concerns. While Rust provides a memory-safe environment for the protocol logic, the underlying building blocks remain subject to C/C++ memory management. We used Valgrind to find and mitigate memory issues occurring due to interaction between Rust and C/C++ codebase space. Our build scripts compile and link the components in our construction. These scripts are designed to be non-destructive and operate within the directories of our artifact. The build files for the FFIs for MAYO (`mayo-c-sys/build.rs`, `mayo-c-rain-sys/build.rs`) call a Makefile that recompiles MAYO with small additional changes. This process computes shared libraries in the respective folders (`mayo-c-sys`, `mayo-c-rain-sys`), which are cleaned up after being combined into a single shared library. Our benchmark script (`bench_all_bs.sh`) modifies the executability of the underlying scripts that build the respective modules in our code. First, the script cleans the build repository of the ZK-proofs (`vole/faest-cpp-tmp/build_debug`) and the build folders of the Blind signatures themselves (`<bs-name>/target`) before running the individual compilations and test executions that are then measured for performance. All executions are local, and we did not identify any ethical concerns relevant regarding the evaluation of this artifact.

A.2.2 How to access

The artifacts of this work are available here: https://github.com/shibammukherjee/pq_blind_signatures, permanent Zenodo concept DOI: https://doi.org/10.5281/zenodo.17979640 pointing to latest version and the Zenodo first version DOI: https://doi.org/10.5281/zenodo.17979641.

A.2.3 Hardware dependencies

We only tested our code on x86 architectures.

A.2.4 Software dependencies

We use and test only on Ubuntu 24.04. The main dependencies are inherited by the underlying building blocks we used. We iterate them, and in braces, we provide the version that we tested. The VOLEitH framework requires `gcc` (>= 14.2.0) or `Clang` (>= 19.1.7), `meson` (>= 1.7.0) and `ninja` (>= 1.12.1).

[1] https://github.com/PQCMayo/MAYO-C
[2] https://github.com/faest-sign/faest-avx
[3] https://github.com/rust-lang/rust-bindgen

The build of MAYO and the RainHash requires `cmake` (>= 3.28.3). The protocol level, i.e., Rust, requires `libclang` to be installed to parse the C/C++ headers for `bindgen`. It is possible to run supplementary tests and benchmarks for some of our constructions with `criterion`[4]. `Criterion` itself requires `gnuplot` for visualizations. These tests can be used to gain additional insights and more confidence in our claims (see Section A.5). For out-of-the-box benchmark runs, use our `Dockfile` built on Ubuntu 25.10, auto-loading all dependencies.

A.2.5 Benchmarks

Our Blind signature size and prove/verify runtime claims can be reproduced by running the `bench_all_bs.sh` script. The script benches each scheme for each security level (NIST Levels 1,3,5) and version (short, fast). Besides our script that runs all benchmarks together, we have scripts to only run the individual constructions: `bench_conservative_deg16_bs.sh`, `bench_optimized_bs.sh`, `bench_rainhash_bs.sh` and `bench_conservative_bs.sh`. The prove/verify runtime may vary depending on the hardware specification of the host machine.

A.3 Set-up

A.3.1 Installation

Installing all software dependencies and verifying their versions are sufficient. We have two options that allow you to verify our claims. You can either follow a native-build for which we provided a manual step-by-step guide explaining how the installation could work. We use a *fresh installation of WSL with Ubuntu 24.04* for this guide, so it follows the minimal steps. The guide can be found in our artifacts in the file: `manual-installation.md`. Alternatively, we provide a dockerfile `Dockerfile` that does the entire setup and the verification can be performed by running the underlying bench scripts. The instructions are also contained as comments in the dockerfile.

A.3.2 Basic Test

Finishing the installation, one runs the bench `bench_all_bs.sh` script. It is, however, possible to run one of the tests from the underlying explicit constructions for a smaller functionality test, e.g., `cd blind-signatures-conservative-deg16 && RUST_MIN_STACK=8388608 cargo test test_and_bench_sign_loop_conservative_128fv1`, which runs a test for SHAKE256-deg16+MAYO-128_f.

[4] https://docs.rs/criterion/latest/criterion/

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): On the 128-bit security level, each of our constructions outperforms current lattice-based proposals (see Table 1) with competitive or improved size. This is proven by our implementations described in Section 8 "Implementation" whose results are illustrated/reported in Table 3 and can be verified with the benchmark script.

A.4.2 Experiments

Our experiments are limited to evaluating the runtime of our algorithms.

(E1): [30 human-minutes[5] + 15 compute-minutes]: This test runs each algorithm described in Table 3 $((\mathsf{Sig}_i)_{i\in\{1,2,3\}}, \mathsf{Ver})$ and reports on their runtime in ms for each security level.
Preparation: Finish the installation as described in Section A.3.1.
Execution: Run the benchmark script `bench_all_bs.sh`.
Results: All of these results are written into the file `bench_log.txt`. The reported runtime numbers, and the signature sizes computed with the VOLEitH signature size formula and checked with total memory allocation, are then displayed in the file. The file has the same format as Table 3, listing first the construction, then the reported performance, and finally the reported sizes of the Blind signatures.

A.5 Notes on Reusability

We have provided some additional benchmarking options using `criterion`. These can also be used for benchmarking with more detailed statistics. We provide FFIs for optimized codebases written in C/C++, which can be reused as they are. These are suffixed with the term `-sys` as is standard for FFIs in Rust. The VOLEitH proof for SHAKE256 and RainHash are embedded in `vole/conservative_bs/owf_proof.inc`. Our ZK-proof use a shared codebase that can be modified to implement other circuit constructions or reuse the same for other additional ZK-proofs over involving, for example, SHAKE256. Similarly, it is also possible to modify the proof of SHAKE256, to, for example, support TurboSHAKE256.

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/usenixsec2026/.

[5] This primarily depends on the installation time, which should hopefully be relatively fast with the dockerfile and the manual installation guide



USENIX Security '26 Artifact Appendix: "Heli: Heavy-Light Private Aggregation"

Ryan Lehmkuhl
MIT

Henry Corrigan-Gibbs
MIT

Emma Dauterman
Stanford

David J. Wu
UT Austin

A Artifact Appendix

A.1 Abstract

This artifact implements Heli, a system that lets a pair of servers collect aggregate statistics about private client-held data with-out learning anything more about any individual client's data. Like prior systems, Heli protects client privacy against a malicious server, protects correctness against misbehaving clients, and supports common statistical functions: average, variance, and more. Heli's innovation is that only one of the servers (the "heavy server") needs to do per-run work proportional to the number of clients; the other server (the "light server") does work sublinear in the number of clients, after a one-time setup phase. As a result, a computationally limited party, such as a low-budget non-profit, could potentially serve as the second server for a Heli deployment with millions of clients.

Heli relies on a new cryptographic primitive, aggregation-only encryption, that allows computing certain restricted functions on many clients' encrypted data. In a deployment with ten million clients, in which the servers privately compute the sum of 32 client-held 1-bit integers, Heli's heavy server does 240,000 core-s of work and the light server does 7 core-ms of work.

A.2 Description & Requirements

All experiments were run on AWS `c7i.*` instances with Ubuntu 22.04 with at least 32 GB of disk storage.

Microbenchmarks were run on a single `c7i.4xlarge` instance (16 vCPUs, 32 GB RAM). They are single-threaded and use $<$ 8 GB of RAM, so can also be run on a smaller machine.

We provide two configurations for reproducing the end-to-end results:

- *Simplified configuration:* Aggregation over 100,000 clients using two `c7i.4xlarge` instances and a `c7i.large` instance. This method takes approximately 20 minutes.
- *Full configuration:* Aggregation over 10,000,000 clients using a `c7i.metal-24xl` instance, one `c7i.4xlarge` instance, and a `c7i.large` instance. This method takes approximately 2 hours.

A.2.1 Security, privacy, and ethical concerns

N/A.

A.2.2 How to access

Our artifact can be accessed via:

- Zenodo at: https://doi.org/10.5281/zenodo.17980903
- Github at: github.com/ryanleh/heli

A.2.3 Hardware dependencies

The artifact requires support for AES-NI CPU intrinsics and at least 8 GB of disk storage.

A.2.4 Software dependencies

Please see the artifact's README.md.

A.2.5 Benchmarks

N/A.

A.3 Set-up

The aggregator and decryptor machines used in the end-to-end evaluation should accept TCP traffic on ports 9000 and 9001.

A.3.1 Installation

Please see the artifact's README.md.

A.3.2 Basic Test

Please see the artifact's README.md.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Heli's computational cost, for the light server, and total server-to-server communication depends only on the number of offline clients in each round. This is proven

by the experiment (E1) described in Section 6.1 and whose results are displayed in Figure 4.

(C2): Heli's computational cost for the heavy server and clients grows linearly with the number of measurements. This is proven by the experiment (E1) described in Section 6.2 and whose results are displayed in Figure 5.

(C3): When privately aggregating 32 1-bit integers, the AWS cost of operating a light server is five orders of magnitude smaller than operating a heavy server. This is proven by the experiment (E2 or E3) described in Section 6.3 and whose results are displayed in Table 2.

Note (updated numbers). After some updates to the implementation, the server numbers reported in Table 2 of the Heli paper have changed slightly. Below is the revised numbers that were confirmed during artifact evaluation:

Number of Measurements		Server aggregation costs: Wall Time (s)	Server aggregation costs: Egress (KB)
$\ell = 1$	*Heavy:*	43.9	3000
	Light:	0.007	0.048
$\ell = 32$	*Heavy:*	873	3000
	Light:	0.008	1
$\ell = 128$	*Heavy:*	3232	3000
	Light:	0.012	4

These changes are reflected in the full version of the paper (https://eprint.iacr.org/2026/059.pdf).

A.4.2 Experiments

(E1): *[Heli Microbenchmarks] [10 human-minutes + 20 compute-minutes]*: This experiment runs the Heli microbenchmarks used in Sections 6.1 and 6.2.
How to: See the artifact's README.md.
Results: The outputted plots should visually match Figures 4 and 5.

(E2): *[Heli Simplified End-to-end Evaluation] [10 human-minutes + 10 compute-minutes]*: This experiment runs a simplified end-to-end Heli workflow that can approximate the server metrics in Table 2. It can be used in place of experiment E3.
How to: See the artifact's README.md for how to run the end-to-end experiments using the "simplified-{1,32,128}.json" config files. The aggregator and client should be run on `c7i.4xlarge` instances or equivalent (in separate regions), and the decryptor should be run on a `c7i.large` instance (in a separate region from the aggregator).
Results: After getting numbers, plug them into the corresponding variables at the top of `benches/estimate_e2e.py` (including details about your machine setup), the result should match Table 2.

(E3): *[Heli End-to-end Evaluation] [10 human-minutes + 2 compute-hours]* This experiment runs the end-to-end Heli workflow to validate the server metrics in Table 2. The aggregator should be run on an `c7i.24xlarge` instance, the decryptor should be run on a `c7i.large` instance (in a separate region from the aggregator), and the client should be run on a `c7i.4xlarge` instance.
How to: See the artifact's README.md for how to run the end-to-end experiments using the "full-{1,32,128}.json" config files.
Results: The results should match the numbers in Table 2.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Quorus: Efficient, Scalable Threshold ML-DSA Signatures from MPC

Alexander Bienstock Leo de Castro Daniel Escudero
Antigoni Polychroniadou Akira Takahashi

JPMC AlgoCRYPT CoE
{firstname}.{lastname}@jpmchase.com

March 18, 2026

A Artifact Appendix

A.1 Abstract

This artifact is a C++ implementation of the online phase of our protocol. The implementation is built atop the Secure Computation Library[1], which implements the channel & network logic for MPC party communication. We provide a simple benchmark file to time the online computation of our protocol for various number of parties at all three ML-DSA security levels. The results of this benchmark are given in Table 6 on page 14 of our paper.

A.2 Description & Requirements

This artifact is a self-contained C++ library that should be compiled to run the benchmark binary. We have tested this library on several version of Linux, including Ubuntu and the AWS EC2 default Linux OS. We have preserved the original `README.md` file from the SCL library, and the build instructions are identical.

Dependencies. There are no major 3rd-party dependencies for this library aside from a compiler, CMake, and some standard C++ libraries. Aside from the basic `build-essentials` like the `make` command, the library should only require

- `g++-13`
- `gmp`
- `catch2`

to compile. We are happy to assist with any issues installing dependencies.

Building the library. To facilitate building the library in the various modes, we have included scripts that run the entire build process. For example, to build the library in "debug" mode, simply run `bash build_debug.bash` from the top directory. This builds the library from scratch and creates a new `build` folder where the binaries are stored. In `debug` mode, the library will perform many correctness checks as it evaluates the test circuits.

Testing the code. To run the complete unit-test suite for this library, simply build the code in `debug` mode and run the binary `./build/test/scl_test`. There is also a binary located in `./build/test_norm_check` that evaluates specific tests for the complete online phase circuit.

Running the benchmark. To run the benchmark of the online phase of our protocol, we recommend to first build the library in `release` mode by running `bash build_release.bash` from the top directory. The benchmark script will still run in `debug` mode, but it will be much slower. Once the library is built, run the binary located at `./build/bench_norm_check`. This binary may take a few minutes to fully complete, and it produces all the data in table 6 in our paper.

A.2.1 Security, privacy, and ethical concerns

This artifact is simply a self-contained C++ program that does not require any special permissions to compile or run. There is no external data used to generate this program; it is simply an implementation of the algorithms described in the paper. We are not aware of any security, privacy, or ethics concerns with this artifact.

A.2.2 How to access

The code is available on Zenodo at this url: https://doi.org/10.5281/zenodo.17888654

ACCESSING THE ARTIFACT: This code was written at JP Morgan Chase, and there are procedures we must follow before publicly releasing any piece of software (even

[1] https://github.com/anderspkd/secure-computation-library

research software) from a regulated financial institution. We have received permission from the AEC chairs to keep the Zenodo files restricted for now. We will make the files public as soon as we have the necessary approvals. In the meantime, please feel free to request access to the Zenodo file. We will accept these requests promptly, and we appreciate your understanding. In order to maintain anonymity, we will accept any request we receive that mentions the USENIX artifact evaluation in the message; feel free to use an obfuscated/anonymized email and a fake name in the request.

A.2.3 Hardware dependencies

None. This artifact should not require any special hardware beyond a normal CPU that can support the standard x86 instruction set.

A.2.4 Software dependencies

The only dependencies our code requires are those inherited from the SCL library as well as g++-13. We give the commands to install each dependency on a Ubuntu Linux OS.

- `gmp`: `sudo apt install libgmp-dev`
- `catch2`: `sudo apt install libcatch2-dev`
- `g++-13`:
  1. `sudo apt install -y software-properties-common`
  2. `sudo add-apt-repository ppa:ubuntu-toolchain-r/test`
  3. `sudo apt update`
  4. `sudo apt install g++-13`

A.2.5 Benchmarks

None. All of the benchmarks in our work are self-contained timings of the online phase of our protocol.

A.3 Set-up

A.3.1 Installation

This library does not need any special permissions to build or run. To build the source code, simply unzip the file stored in the Zenodo entry, install the dependencies, and run the build scripts. We have provided the scripts `build_debug.bash` and `build_release.bash` to facilitate this process. Simple run these scripts from the top directory of the library to build the code. The libraries are placed in the `./build` directory.

A.3.2 Basic Test

To run a simple test, first build the library in `debug` mode by running `bash build_debug.bash`. The unit-tests for this library can be run with the binary `./build/scl_test`. Tests for the online phase of our protocol can be run with `./build/test_norm_check`.

A.4 Evaluation workflow

To run the evaluation, we recommend building the library in `release` mode by running the command `bash build_release.bash` run from the top directory. The benchmark binary can then be run with the command `./bench_norm_check`.

A.4.1 Major Claims

(C1): The script benchmarks the online phase for our protocol, evaluating both the batched comparison function and the batched OR protocol of the online phase, with a batch size matching all three security levels of ML-DSA.

A.4.2 Experiments

(E1): *[Online Phase Benchmark] [2 human-minutes]:* Run the binary `bench_norm_check` to obtain the data in table 6 of our paper. Our benchmarks were run on a single thread of a machine with 32 GB of RAM with an Intel i7 chip running at 2.5 GHz.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Anonymous Tokens with Designated-Reader Metadata Bit

Aisha Tu[†], Meng Jia[‡], Kun He[†*], Jing Chen[†], Ruiying Du[†]
[†]Key Laboratory of Aerospace Information Security and Trusted Computing, Ministry of Education, School of Cyber Science and Engineering, Wuhan University
{aishatu, hekun, chenjing, duraying}@whu.edu.cn
[‡]Department of Computing, The Hong Kong Polytechnic University
mengjia@polyu.edu.hk

A Artifact Appendix

A.1 Abstract

Anonymous tokens with private metadata bit convey hidden signals to verifiers when presented by the user and are under discussion in standardization. Existing solutions only allow the token issuer to read the signals, placing a heavy burden on the issuer and making it challenging to support issuer-hiding, as verifiers must contact the issuer. In this paper, we propose an anonymous token scheme with designated-reader metadata bit, named HinToken, allowing the user to specify an issuer-accepted verifier to read the signal from the token directly. We also extend our scheme to support reader-hiding (HinToken-R) and issuer-hiding (HinToken-I).

This artifact provides implementations and benchmarks of HinToken, HinToken-R, and HinToken-I, and can be run on a single machine with a Rust development environment. In our artifact evaluation, for each construction, we mainly consider the token size and the time costs of key generation, token issuance, and token verification.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

We believe that executing our artifact will not raise any machine security, data privacy, or ethical concerns for evaluators.

A.2.2 How to access

This artifact can be found at https://doi.org/10.5281/zenodo.17946122.

A.2.3 Hardware dependencies

This artifact can be run on most laptops and desktops. We run it on a laptop with Apple M4 Pro chip and 24 GB RAM.

*Corresponding author

A.2.4 Software dependencies

This artifact can be run on a single machine with Rust version 1.88 or later. Any mainstream operating system is compatible (we have tested them on macOS Tahoe, Ubuntu 24.04, and Debian 13; the data in our paper was tested on macOS Tahoe).

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

Run `cargo build --release` to compile the implementations of HinToken, HinToken-R, and HinToken-I.

A.3.2 Basic Test

Run `cargo test --release` to test the implementations of HinToken, HinToken-R, and HinToken-I.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The runtime of IKeyGen, RKeyGen, and Read in HinToken, HinToken-R, and HinToken-I is roughly the same. The runtime of User_0 and Sign in HinToken-R is slower than that in HinToken and HinToken-I, and the runtime of User_1 in HinToken-I is slower than that in HinToken and HinToken-R. Unless the parallel optimization mentioned in Section 7 of our paper fails due to environmental reasons, the runtime of Verify in HinToken-I is roughly the same as that in HinToken and HinToken-R; otherwise, it will be slower than the other two. The running time and token size of our constructions are both worse than those in PMBT and ATHM. All results are shown in Table 1.

A.4.2 Experiments

(E1): Benchmarks of this artifact.

Execution: Run `python3 bench.py` to benchmark the implementations of HinToken, HinToken-R, and HinToken-I. This script will run `cargo bench` to benchmark each implementation and summarize all results in the file `res.md`.

Results: The raw data for each implementation is stored in its respective folder. Taking HinToken as an example, the raw runtime results are displayed in the `index.html` file under the folder `HinToken/target/criterion/report`, including the algorithms IKeyGen (IKeyGen), RKeyGen (RKeyGen), User_0 (user0), Sign (sign0), User_1 (user1), Verify (vrfy), and Read (readbit). The token size is stored in the `_summary_token_size.txt` file under the folder `HinToken/target/criterion`.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: VOID: Defeating Unauthorized Mimicry in Latent Diffusion Models

Chunlin Qiu[1], Ang Li[1], Tianxiao Huang[1], Ruilin Gan[2], Yunjie Ge[3], Shenyi Zhang[3], Huayi Duan[4], Lingchen Zhao[1], Chao Shen[5], Qian Wang[1]*
[1] School of Cyber Science and Engineering, Wuhan University,
[2]School of Computer Science, Wuhan University,
[3]Institute for Math&AI, Wuhan University,
[4]The Hong Kong University of Science and Technology (Guangzhou),
[5]School of Cyber Science and Engineering, Xi'an Jiaotong University

A Artifact Appendix

A.1 Abstract

This final artifact accompanies the *Artifacts Available*, *Artifacts Functional*, and *Results Reproduced* badges awarded to VOID. It includes the VOID implementation, state-of-the-art protection baselines, representative mimicry attacks, evaluation scripts, configuration files, and documentation for running a functional test and a reduced reproduction workflow. The artifact is permanently available under the stable concept DOI https://doi.org/10.5281/zenodo.20233998.

A.2 Description & Requirements

The full evaluation in the paper considers 24 defenses, 10 mimicry/editing methods, and 5 datasets, exceeding the time and compute normally available for artifact evaluation. The AE workflow uses a reduced setting: TI-Dataset [2], two strong image-protection baselines, ACE [12] and Smooth [1], and four mimicry/editing methods: SDEdit/img2img [7], inpainting [9], LoRA [5], and fine-tuning [9]. The following subsections describe access, hardware, software, and benchmark requirements for this evaluation.

A.2.1 Security, privacy, and ethical concerns

The artifact is non-destructive. It does not disable security mechanisms, modify the operating system, or perform network attacks. Experiments run local protection, editing, personalization, and metric-computation pipelines on benchmark images; no private reviewer data is required. Mimicry/editing methods are used only to evaluate defenses. Preparation may download public resources.

*Corresponding author.

A.2.2 How to access

The final artifact is permanently available under the stable concept DOI https://doi.org/10.5281/zenodo.20233998, which resolves to the latest Zenodo version. After downloading and extracting the package, evaluators start from `AE_README.md`, which describes installation, model and dataset preparation, functional testing, reduced reproduction, expected outputs, runtime estimates, and troubleshooting.

A.2.3 Hardware dependencies

The artifact requires an NVIDIA GPU with CUDA support. We recommend at least one GPU with 24 GB of VRAM. When multiple GPUs are available, the scripts can assign different subjects to different GPUs to reduce total runtime; GPU memory is not pooled across devices.

Automatic inference batch-size selection probes the available GPU memory and adjusts the batch size accordingly. On high-memory GPUs, it may select an overly large inference batch size, which can cause a CUDA out-of-memory error. For the safest configuration, evaluators should use `--editing-batch-size 1`. LoRA and fine-tuning already use a training batch size of 1. We also recommend at least 150 GB of free disk space for models, datasets, and generated outputs.

A.2.4 Software dependencies

Required software includes a Linux shell environment, Conda or Anaconda, Python 3.10, and the packages in `requirements.txt`. The setup script creates the conda environment and installs these packages. Network access is needed unless models and datasets are already cached. The model-preparation script downloads or verifies Stable Diffusion v1.5, Stable Diffusion v1.5 Inpainting, BLIP, CLIP ViT-B/32, and SAM ViT-H. Download fallbacks and local-mirror instructions are documented in `AE_README.md`.

A.2.5 Benchmarks

The reduced AE evaluation requires only TI-Dataset [2]. The script `prepare_datasets.sh` with `--dataset ti` restores it under `Datasets/TexturalInversion`. The functional test uses one TI-Dataset subject. The reduced reproduction evaluates clean, VOID, Smooth, and ACE outputs under SDEdit/img2img, inpainting, LoRA, and fine-tuning.

A.3 Set-up

All commands should be run from the repository root after extracting the artifact package.

A.3.1 Installation

Conda or Anaconda must be available before installation. Prepare the environment, models, and dataset as follows:

```
bash scripts/prepare_env.sh
conda activate void-ae
bash scripts/check_gpu.sh
bash scripts/prepare_models.sh
bash scripts/prepare_datasets.sh --dataset ti
```

`prepare_env.sh` creates the `void-ae` environment and installs Python dependencies. `check_gpu.sh` checks CUDA and PyTorch GPU visibility. `prepare_models.sh` downloads or verifies the required models under `Models/`. `prepare_datasets.sh` with `--dataset ti` restores TI-Dataset under `Datasets/TexturalInversion`. After these steps, evaluators can run the basic functionality test below.

A.3.2 Basic Test

Run the basic functionality test with:

```
bash scripts/functional.sh --subject clock
```

The `--subject clock` option selects a deterministic TI-Dataset subject; if omitted, the script selects one subject automatically. A successful run writes outputs to `Results/Functional/TexturalInversion/`, including `manifest.json`, `metrics.json`, protected samples, and editing results.

A.4 Evaluation workflow

This artifact evaluates one functionality claim and two result-reproduction claims in the reduced AE setting. C1 is evaluated by the functional test E1. C2 and C3 are evaluated by the reduced reproduction experiment E2.

A.4.1 Major Claims

(C1): Artifact functionality. The artifact runs end-to-end on one selected subject from TI-Dataset [2]. It verifies environment setup, model loading, protected image generation, SDEdit/img2img [7] editing, and metric computation. This claim is evaluated by E1.

(C2): Protected-image visual utility. VOID generates protected images that remain visually close to the clean originals under the evaluated perturbation budget in the reduced AE setting. This claim is evaluated by E2 using PSNR [6], SSIM [10], and LPIPS [11].

(C3): Mimicry disruption. VOID disrupts unauthorized mimicry across SDEdit/img2img [7], inpainting [9], LoRA [5], and fine-tuning [9]. In the reduced AE setting, VOID is expected to produce more corrupted, noise-like outputs than ACE [12] and Smooth [1] on protected test images. This claim is evaluated by E2 using FID [4], CLIP-I [8], CLIP-S [3], and visual inspection.

A.4.2 Experiments

(E1): Functional test [10 human-minutes + 1 compute-hour]: This experiment evaluates C1 by running the end-to-end functionality test on one TI-Dataset subject.
How to: Run the functional script and inspect the generated metrics and samples under `Results/Functional/TexturalInversion/`.
Preparation: Complete the installation steps above. Ensure that the `void-ae` environment is active, models are prepared, and TI-Dataset is available.
Execution: Run:
`bash scripts/functional.sh --subject clock`
Results: Inspect the output file `Results/Functional/TexturalInversion/metrics.json` and the generated samples. The expected sanity-check result is that original-vs-protected images usually satisfy LPIPS $<$ 0.1, PSNR $\geq$ 32 dB, and SSIM $\geq$ 0.8, while protected SDEdit/img2img outputs lose subject semantics.

(E2): Reduced reproduction [About 10 human-minutes + about 8 compute-hours]: This experiment evaluates C2 and C3 by running the reduced reproduction workflow.
How to: Run the reproduction script and inspect the summary metrics, protected images, editing outputs, and logs under `Results/Reproduce/`, `Results/ProtectResult/`, and `Results/EditResult/`.
Preparation: Complete the installation steps above. The experiment uses TI-Dataset, Stable Diffusion v1.5, and the SD v1.5 inpainting checkpoint. It evaluates clean, VOID, Smooth, and ACE under SDEdit/img2img, inpainting, LoRA, and fine-tuning. When multiple GPUs are available, the script can assign different subjects to different GPUs; their GPU memory is not pooled.
Execution: For the safest GPU-memory configuration, run:
`bash scripts/reproduce.sh --editing-batch-size 1`
Omitting `--editing-batch-size 1` enables automatic inference batch-size selection. If outputs already

exist, recompute metrics only:
`bash scripts/reproduce.sh --only-evaluate`
Results: Main summaries are written under `Results/Reproduce/TexturalInversion/`, including `manifest.json` and `metrics.json`. Protected images are stored under `Results/ProtectResult/`, editing outputs under `Results/EditResult/`, and logs under `Results/Reproduce/`. The artifact reports PSNR, SSIM, LPIPS, FID, CLIP-I, and CLIP-S. Higher PSNR/SSIM, lower LPIPS, higher FID, and lower CLIP-I/CLIP-S indicate the expected trend.
The expected comparative trend corresponds to Table 3 of the paper. Table 3 reports averages over five datasets, whereas E2 evaluates only TI-Dataset. Therefore, exact values may differ; the reduced evaluation focuses on reproducing the same direction of improvement.

A.5 Notes on Reusability

The artifact separates protection generation, image editing/personalization, and metric computation, so these components can be reused independently. Users can extend the workflow by changing the dataset subset, protection method, editing method, perturbation budget, or available GPUs. Generated protection and editing outputs can also be reused for metric-only evaluation with `reproduce.sh --only-evaluate`. The reduced AE scripts provide an entry point for extending the full evaluation framework.

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/usenixsec2026/.

References

[1] Junxi Chen, Junhao Dong, and Xiaohua Xie. Exploring adversarial attacks against latent diffusion model from the perspective of adversarial transferability. *arXiv preprint arXiv:2401.07087*, 2024.

[2] Rinon Gal, Yuval Alaluf, Yuval Atzmon, Or Patashnik, Amit H Bermano, Gal Chechik, and Daniel Cohen-Or. An image is worth one word: Personalizing text-to-image generation using textual inversion. In *International Conference on Learning Representations*, 2022.

[3] Jack Hessel, Ari Holtzman, Maxwell Forbes, Ronan Le Bras, and Yejin Choi. Clipscore: A reference-free evaluation metric for image captioning. *arXiv preprint arXiv:2104.08718*, 2021.

[4] Martin Heusel, Hubert Ramsauer, Thomas Unterthiner, Bernhard Nessler, and Sepp Hochreiter. Gans trained by a two time-scale update rule converge to a local nash equilibrium. *Advances in neural information processing systems*, 30, 2017.

[5] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. In *International Conference on Learning Representations*, 2021.

[6] Quan Huynh-Thu and Mohammed Ghanbari. Scope of validity of psnr in image/video quality assessment. *Electronics letters*, 44(13):800–801, 2008.

[7] Chenlin Meng, Yutong He, Yang Song, Jiaming Song, Jiajun Wu, Jun-Yan Zhu, and Stefano Ermon. Sdedit: Guided image synthesis and editing with stochastic differential equations. *arXiv preprint arXiv:2108.01073*, 2021.

[8] Alec Radford, Jong Wook Kim, Chris Hallacy, Aditya Ramesh, Gabriel Goh, Sandhini Agarwal, Girish Sastry, Amanda Askell, Pamela Mishkin, Jack Clark, et al. Learning transferable visual models from natural language supervision. In *International conference on machine learning*, pages 8748–8763. PmLR, 2021.

[9] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE/CVF conference on computer vision and pattern recognition*, pages 10684–10695, 2022.

[10] Zhou Wang, Alan C Bovik, Hamid R Sheikh, and Eero P Simoncelli. Image quality assessment: from error visibility to structural similarity. *IEEE transactions on image processing*, 13(4):600–612, 2004.

[11] Richard Zhang, Phillip Isola, Alexei A Efros, Eli Shechtman, and Oliver Wang. The unreasonable effectiveness of deep features as a perceptual metric. In *Proceedings of the IEEE conference on computer vision and pattern recognition*, pages 586–595, 2018.

[12] Boyang Zheng, Chumeng Liang, and Xiaoyu Wu. Targeted attack improves protection against unauthorized diffusion customization. *arXiv preprint arXiv:2310.04687*, 2023.





USENIX Security '26 Artifact Appendix: From Easy to Hard++: Promoting Differentially Private Image Synthesis Through Spatial-Frequency Curriculum

Chen Gong[†]
University of Virginia

Kecen Li[†]
University of Virginia

Zinan Lin
Microsoft Research

Tianhao Wang
University of Virginia

A Artifact Appendix

A.1 Abstract

This is the official implementation document of the Usenix Security paper, "From Easy to Hard++: Promoting Differentially Private Image Synthesis Through Time-Frequency Curriculum." In the `README.md` files of the repository[1], we provide a clear, point-by-point explanation that maps each table and figure to the corresponding code needed to generate its results. We provide a detailed description of how our repository matches each table and figure as follows.

A.2 Description & Requirements

Each table/figure is explicitly mapped in README.md, so reviewers can locate and run the exact implementation for any reported result. We isolate parameter configurations from implementations. Users can adjust them via "config" files.

A.2.1 Security, privacy, and ethical concerns

Our artifact is a research implementation for differentially private (DP) image synthesis and poses no security risks to evaluators' machines.

A.2.2 How to access

The DOI for the artifacts is on Zenodo.[2]

A.2.3 Hardware dependencies

- **GPU**: 4 NVIDIA GPUs with CUDA support (minimum: each GPU with 32GB VRAM).
- **CPU**: Multi-core CPU (8+ cores recommended) for data preprocessing and evaluation metrics.
- **Memory**: 32GB+ system RAM recommended for handling large batches and FID computation.
- **Storage**: 10GB free disk space for datasets, checkpoints, and generated samples.

A.2.4 Software dependencies

- **OS**: Linux
- **Python**: 3.9
- **Core packages** are listed in requirements.txt.
- **CUDA**: 11.8 or 12.1 (matching your PyTorch build)
- **Optional**: Docker (we provide a `Dockerfile` for containerized execution)

A.2.5 Benchmarks

- **Datasets:** all public, auto-download scripts are provided in `data_preparation.sh`
- **Evaluation metrics:** FID, classification accuracy.

A.3 Set-up

A.3.1 Installation

We provide a straightforward environment setup. The user only needs to run "bash install.sh" to install our environment in minutes, supporting availability and functionality assessments.

A.3.2 Basic Test

After activating the environment, execute: "python run.py setup.n_gpus_per_node=4 -m DP-FETA-Pro -dn mnist_28 -e 1.0 eval.mode=val pretrain.n_epochs_time=1 pretrain.n_epochs_freq=1 train.n_epochs=1 gen.data_num=100". Quantitative results will be logged in `exp/feta-pro/<exp_name>/stdout.txt`.

[1] https://github.com/2019ChenGong/Feta-Pro
[2] https://doi.org/10.5281/zenodo.17836341

A.4 Evaluation workflow

A.4.1 Major Claims

We tackle a central challenge in privacy-preserving ML: achieving a strong balance between utility and privacy without depending on potentially risky public images. DP-FETA-Pro introduces a novel framework that integrates spatial- and frequency-domain representations, offering new insights into the design of privacy-preserving generative models. DP-FETA-Pro achieves better FID and Acc than the state-of-the-art method. This is proven by the experiments.

A.4.2 Experiments

Environment: We provide a straightforward environment setup. The user only needs to run "bash install.sh" to install our environment in minutes, supporting availability and functionality assessments.

Data Preparation: This artifact provides a concrete data preparation process. All datasets used are publicly available and widely used in the community to avoid ethical concerns. The user should run "bash data_preparation.sh" to prepare all the studied datasets. Loading all datasets is time-consuming. For quick evaluation, we recommend loading only `MNIST` and `Fashion-MNIST`, using "bash data_preparation_quick.sh".

Main Contributions: We present how to match each table and figure in the main contributions of our paper as follows.

(1) *Reproducing Main Contributions (RQ1).* The strengths of our methods compared to baselines are reported in Table 4, Figure 3, and Figure 4. To train the DP-FETA-Pro synthesizer on the `MNIST` dataset with a privacy budget $\varepsilon = 1.0$ (as shown in Table 3), execute: "python run.py setup.n_gpus_per_node=4 –method DP-FETA-Pro –data_name mnist_28 -e 1.0 eval.mode=val".

To reproduce baseline results using the `DPImageBench` suite, modify the `-m` (method) and `-ed` (experiment description) flags. Supported baselines include `DP-MERF`, `DP-NTK`, `DP-Kernel`, `DPDM`, `DP-GAN`, `GS-WGAN`, and `DP-FETA`. Example for `DPDM`: "python run.py setup.n_gpus_per_node=4 setup.master_port=6662 -m DPDM -dn mnist_28 -e 1.0 -ed dpdm eval.mode=val".

Quantitative results of our experiments are logged in `exp/pdp-diffusion/<exp_name>/stdout.txt`. For Figure 3 (Visualizations) and Figure 4 (FID Curves), use: "python plot/visualization.py" for image quality visualization; and "python plot/fid_curve.py" for convergence analysis.

(2) *Ablation Studies (RQ2):* RQ2 investigates the benefits of frequency features and the auxiliary generator (Table 5 and Figure 5). Users can select frequency learning invariants via `pretrain.mode` as follows.

- `FETA-Pro_ft`: Prioritizes spatial then frequency (`freq_time`).
- `FETA-Pro_mix`: Simultaneous learning (`mix`).
- `FETA-Pro_f`: Frequency-only warm-up (`freq`).

Example command for "*mix*" mode: "python run.py pretrain.mode=mix -m DP-FETA-Pro -dn mnist_28 -e 1.0"

To compare with `No-Auxiliary` and `DM-Auxiliary` (Table 4), set `pretrain.mode` to `no_auxiliary` or `dm_auxiliary` respectively.

(3) *Privacy Budget Allocation (RQ3).* To reproduce the experiments of privacy allocation (Figures 6, 7, and Table 11): Adjust noise scales via `train.sigma_freq` (frequency domain) and `train.sigma_time` (spatial domain) as, "python run.py –method DP-FETA-Pro -e 1.0 train.sigma_freq=26.6 train.sigma_time=20". For Figure 7, users can adjust `-e` to run FETA-Pro with varying privacy budget.

Table 11 provides the RDP cost ratios. These can be computed directly using our analytical tool: "python cal_privacy.py –method DP-FETA-Pro -e 1.0 train.sigma_freq=26.6 train.sigma_time=20".

(4) *Extended Analysis and Discussion (results in Section 6.1 and Section 6.2).* To facilitate the experiments discussed in the extended analysis sections of the paper, we provide specialized scripts for non-private baselines and transfer learning scenarios. To establish a performance upper bound, the classification algorithm can be tested on sensitive images without DP protections is "python ./scripts/test_classifier.py –data_name mnist_28 –epsilon 10.0 -ed no-dp-mnist_28".

For experiments involving warm-up training with public images (as discussed in the Discussion section), set the pre-training mode to `time_freq`: "python run.py setup.n_gpus_per_node=3 pretrain.mode=time_freq –method PDP-Diffusion –data_name mnist_28 -e 10.0".

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: CompLeak: Deep Learning Model Compression Exacerbates Privacy Leakage

Na Li[1*] Yansong Gao[2*] Hongsheng Hu[3] Boyu Kuang[1 †] Anmin Fu[1 4 †]

[1]*School of Cyber Science and Engineering, Nanjing University of Science and Technology, China.*
[2]*School of Cyber Science and Engineering, Southeast University, China.*
[3]*School of Computer Science, Shanghai Jiao Tong University, China.*
[4]*School of Computer Science and Engineering, Nanjing University of Science and Technology, China.*

A Artifact Appendix

A.1 Abstract

This artifact provides an implementation of CompLeak, a privacy risk evaluation framework that investigates how model compression amplifies membership inference attacks (MIA). CompLeak supports three widely used compression techniques, including pruning, quantization, and weight clustering, and examines three attack scenarios—CompLeak $_{NR}$, CompLeak $_{SR}$, and CompLeak $_{MR}$—differing in the information accessible to the adversary.

The artifact comprises scripts for training original models, performing model compression, and executing CompLeak $_{NR}$, CompLeak $_{SR}$, and CompLeak $_{MR}$. It supports experiments with ResNet-18 on CIFAR-10. Please refer to the contained README.md for a detailed overview of the artifact contents and their organization.

A.2 Description & Requirements

This section describes the experimental setup, including hardware and software requirements, as well as the benchmarks used to generate the results.

A.2.1 Security, privacy, and ethical concerns

Executing this artifact raises no security, privacy, or ethical issues. All experiments use public datasets and widely used model architectures.

A.2.2 How to access

Our artifact is publicly available at: https://figshare.com/articles/software/CompLeak/30904289

We also provide the concept DOI of the artifact: https://doi.org/10.6084/m9.figshare.30904289

A.2.3 Hardware dependencies

We recommend an NVIDIA GeForce RTX 3080 or a more powerful GPU, as runtime depends on GPU performance. There are no strict CPU or memory requirements.

A.2.4 Software dependencies

All dependencies are listed in requirements.txt, with installation instructions provided in Section A.3.1.

A.2.5 Benchmarks

Detailed instructions are provided in Section A.3.1.

A.3 Set-up

This section describes how to install and configure the environment required to run the artifact.

A.3.1 Installation

The installation process includes dependency installation, data preparation, original model training, and model compression.

Dependency installation. Install all dependencies specified in requirements.txt:

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

Data preparation. The CIFAR-10 will be automatically downloaded to the data/datasets/ folder when the code is executed.

Original model training. Train the original model by running follow command:

*The authors contribute equally to this paper.
†Corresponding authors.

```
python train.py --dataset_name cifar10
--model_name resnet18 --num_cls 10 --lr 0.1
--epochs 100 --batch_size 128
```

Model compression. To compress the original model, follow these steps to obtain pruned models (at 60%, 70%, 80%, and 90% levels), an int-8 quantized model, and weight-clustered models (with cluster centers of 4, 8, and 16).

- **Pruning.** Use `-prune_sparsity` to set the pruning ratio i.e., 0.6, 0.7, 0.8, 0.9 for 60%, 70%, 80% and 90% pruning:

  ```
  python pruning/prune.py --dataset_name cifar10
  --model_name resnet18 --num_cls 10 --lr 0.001
  --prune_epochs 30 --prune_sparsity 0.6
  ```

- **Quantization.** Run:

  ```
  python quantization/qat.py  --degree int8
  --model_name resnet18 --num_cls 10 --lr 0.001
  --quantized_epochs 30 --dataset_name cifar10
  ```

  Weight Clustering. Use `-bit` to control cluster levels i.e., 2, 3, 4 for 4, 8, 16 cluster centers:

  ```
  python clustering/clustering.py
  --model_name resnet18 --num_cls 10
  --lr 0.00001 --epochs 30 --bit 2
  --dataset_name cifar10
  ```

At this point, the following models have been trained:

- One original model (full precision) in `results/`
- Multiple compressed models with different compression techniques and levels:
  - Pruned models with 60%, 70%, 80%, and 90% sparsity in `results_pruning/`
  - An INT-8 quantized model in `results_quantization/`
  - Clustered models with 4, 8, and 16 cluster centers in `results_clustering/`

A.3.2 Basic Test

You can run the following command to check that required dependencies are installed correctly. If the process displays 'Start Membership Inference Attacks,' it indicates that all dependencies are installed correctly.

```
python CompLeakSR.py --compress_degree 0.9
--dataset_name cifar10 --num_cls 10
--batch_size 128 --original --compress
--model_name resnet18 --compress_name pruning
```

A.4 Evaluation workflow

This section presents the key claims of the paper long with the experiments that support them.

A.4.1 Major Claims

The key claims presented in our paper are as follows:

(C1): `CompLeak` $_{\text{NR}}$ applies six existing MIAs to audit the privacy vulnerabilities of the original model and its per compressed variants. The results show that highly compressed models are generally less vulnerable to MIA than the original model, whereas other compressed models—e.g., pruning with low sparsity, int 8-bit quantization—exhibit privacy leakage comparable to the original model. This is proven by the experiment (E1), with reported in Section 4.

(C2): `CompLeak` $_{\text{SR}}$ leverages the compressed model as a reference, combining information from both the original model and one compressed model. The results reveal that compression operations indeed jeopardize privacy, outperforming `CompLeak` $_{\text{NR}}$'s performance on the original or compressed model. This is proven by the experiment (E2), with reported in Section 5.

(C3): `CompLeak` $_{\text{MR}}$ leverages the multiple compressed model as the reference. The results show that aggregating leakage from multiple compressed models further amplifies the privacy leakage. This is proven by the experiment (E3), with reported in Section 6.

A.4.2 Experiments

The key claims are supported by the following experiments:

(E1): [`CompLeak` $_{\text{NR}}$] [20 human-minutes + 3 compute-hour]: `CompLeak` $_{\text{NR}}$ (six existing MIAs) attacks each model individually, including the original and each compressed variant (pruned with 60%, 70%, 80%, 90%, int8-bit quantized, and weight clustered with 4, 8, 16 centroids.

Preparation: No further setup is needed once the data preparation, model training, and compression described in A.3.1 are complete.

Execution: To run the experiment, execute the `CompLeakNR.py`. First, apply `CompLeak` $_{\text{NR}}$ to the original model:

```
python CompLeakNR.py --model_name resnet18
--dataset_name cifar10 --num_cls 10
--batch_size 128 --original
```

Next, apply `CompLeak` $_{\text{NR}}$ to the compressed model.

Pruning:

Pruning ratio can be adjusted via `-compress_degree` (0.6, 0.7, 0.8, 0.9) for 60%, 70%, 80%, and 90% pruning.

```
python CompLeakNR.py --compress_degree 0.9
--dataset_name cifar10 --num_cls 10
--batch_size 128 --compress
```

`--compress_name pruning --model_name resnet18`

Quantization:

```
python CompLeakNR.py --compress_degree int8
--dataset_name cifar10 --num_cls 10
--batch_size 128 --compress
--compress_name quantization
--model_name resnet18
```

Weight clustering:

The number of centroids can be adjusted via `-compress_degree` to 4, 8, or 16.

```
python CompLeakNR.py --compress_degree 4
--dataset_name cifar10 --num_cls 10
--batch_size 128 --compress
--compress_name clustering
--model_name resnet18
```

Results: Each run outputs results for six MIAs. For each attack, we report ACC, AUC, and TPR at 0.1% FPR. Only highly compressed models, e.g., 90% pruning, show less vulnerability to MIAs compared to the original model (**C1**).

(E2): [$\texttt{CompLeak}_{\text{SR}}$] [20 human-minutes + 1 compute-hour + 75GB disk]: $\texttt{CompLeak}_{\text{SR}}$ combining information from both the original model and one compressed model, reveal that compression operations indeed jeopardize privacy (pruned with 60%, 70%, 80%, 90%, int8-bit quantized, and weight clustered with 4, 8, 16 centroids).

Preparation: No further setup is needed once the data preparation, model training, and compression described in A.3.1 are complete.

Execution: To run the experiment, execute the `CompLeakSR.py`. The `-compress_degree` parameter controls both the pruning ratio (0.6, 0.7, 0.8, and 0.9 corresponding to 60%, 70%, 80%, and 90% pruning) and the number of centroids (4, 8, or 16).

Validate pruning operation:

```
python CompLeakSR.py --compress_degree 0.9
--dataset_name cifar10 --num_cls 10
--batch_size 128 --original --compress
--model_name resnet18
--compress_name pruning
```

Validate quantization operation:

```
python CompLeakSR.py --compress_degree int8
--dataset_name cifar10 --num_cls 10
--batch_size 128 --original --compress
--model_name resnet18
--compress_name quantization
```

Validate weight clustering operation:

```
python CompLeakSR.py --compress_degree 4
--dataset_name cifar10 --num_cls 10
--batch_size 128 --original --compress
--model_name resnet18
--compress_name clustering
```

Results: We report ACC, AUC, and TPR at 0.1% FPR. Results on pruned models (60%, 70%, 80%, and 90%), int8-bit quantized model, and weight clustered models with 4, 8, and 16 centroids show that $\texttt{CompLeak}_{\text{SR}}$ outperforms $\texttt{CompLeak}_{\text{NR}}$'s performance on a single original/compressed model (C2).

(E3): [$\texttt{CompLeak}_{\text{MR}}$] [1 human-minutes + 3 compute-hour + 75GB disk]: $\texttt{CompLeak}_{\text{MR}}$ leverages the multiple compressed model as the reference. The results show that aggregating leakage from multiple compressed models further amplifies the privacy leakage.

Preparation: No additional setup is required once the data preparation, model training, and compression steps described in Section A.3.1 are completed, and $\texttt{CompLeak}_{\text{SR}}$ has been applied to all compressed models (a total of eight) in E2.

Execution: Open `CompLeakMR.ipynb` in Jupyter and execute all cells ("Run", "Run All Cells").

Results: The results of $\texttt{CompLeak}_{\text{MR}}$, presented in cell 19 of `CompLeakMR.ipynb`, show that aggregating leakage from multiple compressed models further amplifies privacy leakage compared to $\texttt{CompLeak}_{\text{SR}}$ (C3).

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: From Assistance to Autonomy: An Empirical Study of AI Use in a Live Capture-the-Flag (CTF) Competition

Tingxuan Tang[1], Nicolas Janis[1], Kalyn Asher Montague[1], Kevin Eykholt[2], Dhilung Kirat[2], Youngja Park[2], Jiyong Jang[2], Adwait Nadkarni[1], Yue Xiao[1]
[1]William & Mary, [2]IBM Research

A Artifact Appendix

A.1 Abstract

This artifact provides the complete set of materials needed to reproduce our main claims. The artifact contains five components: (1) the 17 challenges in TribeCTF 2025. The challenge set is released in three formats to support experimental conditions; (2) the full code of CTFriend, the agentic, MCP-based chat assistant we provided to human participants during the live competition; (3) autonomous agent run logs from three agent frameworks (Claude Code, NYU Autonomous Framework, and Cybench), each evaluated with three Claude model variants across all 17 challenges; Two scripts are included to generate score tracking figure and performance comparison table; (4) the qualitative analysis codebooks used to code human-AI interaction behavior and attitude shifts; and (5) the pre- and post-CTF survey instruments distributed to human participants. Each components have their own `README.md` file with more details.

A.2 Description & Requirements

Survey instruments, qualitative analysis codebooks, autonomous agent logs, and challenges are readable directly.

CTFriend is a website-based chatbot with CTF domain knowledge supported. In order to reproduce it, the only requirement is to have a standard computer with Docker and Docker Compose.

Due to the nondeterminism in LLMs, it would be challenging to reproduce the autonomous agent performance; thus, we provide the original run logs of all agents during competition, and there are two scripts used to generate Figure 4 and Table 2 in RQ3 in the paper.

A.2.1 Security, privacy, and ethical concerns

The artifact poses minimal risk to evaluators. The challenge files, logs, codebooks, and surveys are static, non-executable data that can be reviewed safely. ***CTFriend*** is a chatbot-style AI assistant working as a website. Evaluators should follow the instruction to deploy and run it on localhost. All interaction data will be stored locally and can only be accessed by users; thus, the deployment and running process has no specific risks.

A.2.2 How to access

The artifact is available at https://doi.org/10.5281/zenodo.20309511. The only step is to download and unzip it. All components have their specific folders.

A.2.3 Hardware dependencies

The artifact can be tested on a commodity x86-64 (or Apple Silicon) machine capable of running Docker and Docker Compose. The complete artifact is about 7 GB, so roughly 4 GB RAM and 15 GB storage are sufficient.

A.2.4 Software dependencies

For the data components (challenges, autonomous agent logs, qualitative codebooks, survey instruments), only a reader or editor able to open .docx, .json, and .yaml/.yml files are required; no compilation is needed.

CTFriend is fully containerized, and its only host prerequisites are Docker and Docker Compose (installation: https://docs.docker.com/compose/install/). All other dependencies are pulled or built automatically when building Docker containers, including the application stack (Python, FastMCP/Streamlit, a Node.js CyberChef backend, and a RAG knowledge-base MCP server) and the supporting services.

Dependencies for two scripts are listed in the `/Autonomous_Agent_Logs/requirement.txt` file, and Python 3.9+ is necessary.

A.2.5 Benchmarks

The competition/experiment uses the 17 TribeCTF 2025 challenges included in this artifact (`/Challenges`) as the eval-

uation benchmark. These challenges were newly designed by a third-party security company and had never been used before. They are released here in three formats, provided to human players/Claude Code, NYU agent, and Cybench agent correspondingly, to match the competition and experiment conditions.

A.3 Set-up

The artifact has both inspectable parts and executable parts; the setup effort depends on which part is being evaluated. After downloading and unzipping the artifact file,

• *Inspectable components:* require no installation; evaluators only need to open the files in `/Challenges`, `/Autonomous_Agent_Logs`, `/Qualitative_Anaysis_Codebook`, and `/Survey_Instrument` with a PDF reader and a text/.docx/.json/.xlsx/.yaml viewer.

• *CTFriend:* is fully containerized and brought up with Docker Compose.

•*Agent-performance script:* is a small standalone Python tool that reproduces the RQ3 results from the provided logs.

A.3.1 Installation

1. **Enter CTFriend folder**

```
cd /CompleteArtifact/CTFriend
```

2. **Setup environment variables**

```
# Get a local environment file
cp .env.example .env

# Edit environment varibles
nano .env
```

POSTGRES_PASSWORD and TOKEN_SECRET_KEY only need to be changed in live deployment. For evaluation, the defaults are fine. Please set the ANTHROPIC_API_KEY to your own key.

3. **Install Docker containers**

```
# Insure docker is running
docker-compose up --build
```

4. **Generating User Tokens**

CTFriend works on whitelisting tokens, where each user has their own individual token to ID them and their chats within the system. To create this token, use the following instructions.

```
# Log in to the ctfriend container
docker exec -it ctfriend bash

cd app
python3 token_manager.py whitelist [email]
# NOTE: Any email works here. For testing purposes, feel free to use test@test.com
```

Then you will get your specific user token. which will be used later.

5. **Configure Grafana**

• Open your web browser and go to `http://localhost:3000`.

• Log in with "username: `admin`" and "password: `admin`".

6. **Prometheus Setup**

• *Add the Prometheus Data Source:* Go to "Connections" -> "Data Sources" -> "Add new data source" -> Select "Prometheus".

• For the "Prometheus server URL", enter `http://prometheus:9090`.

• Click "Save & Test".

7. **PostgreSQL Setup**

• *Add the PostgreSQL Data Source:* Go to "Connections" -> "Data Sources" -> "Add new data source" -> Select "PostgreSQL".

• *Use the following connection details:* Name: `postgres_db`, Host: `db:5432`, Database name: `metrics`, User: `postgres`, Password: `change-me-to-a-secure-password` or `<password>` `set in .env`, TLS/SSL Mode: `disable`, Version: `15`, Timescale DB: `True`.

• Click "Save & Test". You might need to give Grafana's data source a specific UID like postgres_db if you want it to automatically link with the chat data dashboard.

8. **Setup Dashboards**

• *Import the Dashboards:* Go to "Dashboards" -> "New" -> "Import".

• Upload the `system_monitoring_dashboard.json` file. Select the "Prometheus" data source when prompted.

• Repeat the process for the `chat_data_dashboard.json` file, selecting the `PostgreSQL` data source.

9. **Start CTFriend**

• Go to `http://localhost:8501/ctfriend`.

• Input your specific login token generated in **step 4**.

• Select `Anthropic` as provider and `claude-opus-4-1-20250805` as model.

• Now ***CTfriend*** is ready.

A.3.2 Functionality Test

The tests are designed for CTFriend.
Test 1: Authenticated chat.

• Send "Hello, can you confirm you are working?"
Expected: CTFriend replied normally. An invalid/empty token instead returns "Please enter a valid token.", confirming access control is active.

Test 2: MCP tool use.

• Send "Use CyberChef to Base64-decode: Q1RGcmllbmQ"
Expected: the assistant calls the CyberChef MCP tool and returns CTFriend, confirming cyberchef_api and cyberchef_backend are reachable.

Test 3: RAG knowledge base.

• Send "What is the $MFT in NTFS forensics?"
Expected: a grounded answer drawn from the bundled knowledge base via rag_server.

A.4 Evaluation workflow

The study addresses three research questions: **RQ1** — how participants' attitudes toward AI assistance shift before vs. after the competition (based on survey responses); **RQ2** — how humans interact with AI assistance during problem-solving (qualitative coding of CTFriend interaction data); and **RQ3** — how autonomous agents perform on the same challenges across frameworks and models.

Since RQ1 and RQ2 derive from real human responses and interaction, to protect participants' privacy as presented in the institutional IRB, the corresponding claims are validated by inspecting the provided instruments and codebooks. The RQ3 claim is re-executable: the provided logs and workbook can be re-aggregated with two scripts.

A.4.1 Major Claims

(C1): An advanced agent outperforms most human teams while requiring only about 1/5 of the cumulative runtime. Meanwhile, model capability ultimately sets the ceiling. Even strong tooling and architecture cannot compensate for a weak base model, and under weaker models the different frameworks largely converge to similarly low performance.

(C2): *Crypto and forensics are comparatively easier for the agent teams than for humans, whereas reverse engineering remains harder for agents and favors human expertise; these mismatches suggest that human and AI strengths are complementary.*

A.4.2 Experiments

(E1): [Score Tracking] [3 human-minutes + 3 compute-minutes + 5GB disk]: This experiment is used to generate Figure 4 (score tracking) in the paper based on original autonomous agent logs, manual-recorded work records, and human score tracking records exported by the TribeCTF 2025 committee.
How to: go to the `Autonomous_Agent_Logs` folder and run the script and observe the output figure
Preparation: prepare a new terminal window with pre-installed Python and install `requirements.txt`

```
cd /CompleteArtifact/Autonomous_
    Agent_Logs

# install required dependencies
pip3 install -r requirements.txt
```

Execution:

```
# run the figure generation script
python3 scoretracking.py
```

Results: The output should be the same as Figure 4 in the paper, where we can observe that the most advanced model, PA(S), completed the competition in about 4h (Agent Scoreboard), but human teams usually took more than 20h (Human Scoreboard). In Agent Scoreboard, for each agent (colored in gradient color group), sonnet-4.5 (S) always shows the best performance while Haiku-3.5(H) falls behind.

(E2): [Performance Comparison] [3 human-minutes + 3 compute-minutes + 5GB disk]: This experiment is used to generate Table 2 in the paper based on original autonomous agent logs and manual-recorded work records
How to: similarly, run the `statistics.py` file and observe the output table
Preparation: prepare a new terminal window with pre-installed Python and install required dependencies (should be done in E1)
Execution:

```
# run the figure generation script
python3 statistics.py
```

Results: The results should be the same as Table 2 in the paper. It shows the success rate of each agent by categories. The success rate of Crypto and forensics for most agents with Sonnet-4.5 and Opus-4.1 is more than 50%, and even 100%; however, the success rate of reverse engineering is mostly 0%, and only a few agents are able to achieve 33% or 67% with advanced models.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Chameleon Channels: Measuring YouTube Accounts Repurposed for Deception and Profit

Alejandro Cuevas
Carnegie Mellon University

Manoel Horta Ribeiro
Princeton University

Nicolas Christin
Carnegie Mellon University

A Artifact Appendix

A.1 Abstract

This artifact is a reproducibility package for the paper *"Chameleon Channels: Measuring YouTube Accounts Repurposed for Deception and Profit."* It includes (1) an R script to reproduce the paper's DSL-based modeling results and (2) a Python analysis pipeline to reproduce the paper's main figures (pre/post change, Sankey flow visualization, and survivability / Kaplan–Meier analysis), as well as the combined descriptive statistics table. The artifact is designed to run end-to-end on a standard Linux/macOS environment with R and Python installed, using the curated data files distributed via a restricted Zenodo repository.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

The artifact uses datasets derived from YouTube channels and Fameswap listings. Distribution is governed by YouTube's Data API policy and Fameswap's terms of service, which may inhibit public release. As a result, the replication data are distributed via a **restricted-access** Zenodo repository.

Evaluators should: (i) treat the provided data as confidential and store it locally on an access-controlled machine, (ii) avoid redistribution of the datasets and any access tokens, (iii) use the data only for artifact evaluation / academic reproduction purposes as permitted, and (iv) follow any additional conditions specified in the Zenodo record and/or accompanying documentation.

The dataset includes channel metadata such as handles and handle changes (public identifiers); evaluators should avoid re-publication of raw identifiers beyond what is required for reproducing the paper's results.

A.2.2 How to access

The data and reproduction code are available in a restricted Zenodo repository: https://doi.org/10.5281/zenodo.17955052. When requesting access, please indicate what your intended usage will be.

A.2.3 Hardware dependencies

None. A commodity laptop/desktop is sufficient.

A.2.4 Software dependencies

OS: Linux or macOS (the artifact uses Bash scripts).
R: version 4.0+ recommended.
R packages: `dsl` v.0.1.0 (see installation instructions from Egami et al. for the R package.).
Python: version 3.9+ recommended.
Python packages: installed via `requirements.txt`: `pandas`, `numpy`, `matplotlib`, `seaborn`, `plotly`, `lifelines`, `scipy`, `pyarrow`.
Bash (for running the setup script).

A.2.5 Benchmarks

The artifact reproduces results using curated datasets distributed in the restricted Zenodo repository. The main samples considered in the study are described in Appendix B.

To replicate the main findings, the artifact requires the labeled/categorized repurposed channels from Fameswap (n=1,024), labeled/categorized repurposed channels from Social Blade (n=1,047), and the Social Blade baseline sample (n=2,972), all restricted to channels with >1,000 subscribers. These data include handle changes per channel and are sufficient to reproduce: Figure 6, Figure 7, Figure 8, and Table 1.

Concretely, the repository expects the following files under `./data/`:

- `socialblade_v_baseline.csv`
- `fameswap_v_baseline.csv`
- `control_event_study.parquet`
- `treatment_event_study.parquet`
- `kaplan_data.parquet`
- `sankey_data.parquet`

A.3 Set-up

A.3.1 Installation

Obtain the artifact bundle from the restricted Zenodo repository (links above) and unpack it.

From the artifact root, create required directories:

```
mkdir -p data output
```

Place the required CSV/parquet files (listed in § Description & Requirements) into `./data/`.

Install the R dependency. Refer to the R package documentation for the latest installation instructions. At this time, the recommended installation method for (`dsl`) is:

```
if(!require(devtools))
 ↪  install.packages("devtools")
library(devtools)
install_github("naoki-egami/dsl",
 ↪  dependencies = TRUE)
library(dsl)
```

(Optional but recommended) Ensure Python can create a virtual environment and that `pip` is up to date.

After these steps, the evaluator should be able to run the basic tests below.

A.3.2 Basic Test

This basic test checks: (i) required input files exist, (ii) R can load `dsl`, and (iii) Python dependencies can be imported.

(1) Verify required data files exist From the artifact root:

```
ls -lh data/
```

Expected outcome: the directory listing includes all six required files (`*.csv` and `*.parquet`) named in § Benchmarks.

(2) Verify R dependency loads

```
Rscript -e 'library(dsl); cat("dsl: OK\n")'
```

Expected outcome: prints `dsl: OK`.

(3) Verify Python dependencies import

```
python3 -c "import pandas, numpy, matplotlib,
 ↪  plotly, lifelines, scipy, pyarrow;
 ↪  print('python: OK')"
```

Expected outcome: prints `python: OK`.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): The artifact reproduces the paper's pre/post repurposing analysis results shown in Figure 6 via Experiment (E1).

(C2): The artifact reproduces the paper's channel-handle/topic transition visualization shown in Figure 8 via Experiment (E2).

(C3): The artifact reproduces the paper's survivability analysis of channels for sale (Kaplan–Meier style) shown in Figure 7 via Experiment (E3).

(C4): The artifact reproduces the paper's combined descriptive statistics reported in Table 1 via Experiments (E4).

A.4.2 Experiments

There are four main experiments that this artifact supports. E1-3 are Python-based, and E4 is R-based. We provide E0 as a convenient script to execute E1-3.

(E0): *End-to-end Python pipeline (convenience runner) [10–20 human-minutes + 10–30 compute-minutes + ∼1–5GB disk; approximate]*
How to: Use the provided Bash script to create an isolated Python environment and run all Python analyses (E1 to E3).
Preparation: Ensure `requirements.txt` is present and all required `./data/` files exist.
Execution: `chmod +x run_python_exps.sh` and then `./run_python_exps.sh`
Results: The script should (i) create a virtual environment, (ii) install dependencies, (iii) verify required data files exist, and (iv) run `kaplan.py`, `pre_post_change.py`, and `sankey.py`.
Expected outcome: all scripts complete without error and populate `./output/` with the files needed for Figure 6, Figure 8, and Figure 7.

(E1): *Pre/Post Change (Figure 6) [15–30 human-minutes + 5–20 compute-minutes + ∼1–5GB disk; approximate]*
How to: Run the Python pre/post analysis script over the provided event-study parquet files.
Preparation: Ensure `control_event_study.parquet` and `treatment_event_study.parquet` exist and are in the `data` folder. Ensure Python dependencies are installed.
Execution: `python3 pre_post_change.py`
Results: The script should write one figure file into `./output/` corresponding to Figure 6.
Expected outcome: output files are created and are non-empty; the plot matches the paper qualitatively (same direction and shape of the effect, with the same axes/structure).

(E2): *Sankey Visualization (Figure 8) [10–20 human-minutes + 2–10 compute-minutes + ∼1GB disk; approximate]*
How to: Generate the Sankey diagram from the provided `sankey_data.parquet`.
Preparation: Ensure `sankey_data.parquet` exists, is placed in the `data` folder, and Python dependencies are installed.
Execution: `python3 sankey.py`
Results: The script should write a Sankey visualization into `./output/` (commonly `.html` for Plotly, or an exported static image if configured).

Expected outcome: the file opens correctly and visually matches Figure 8 in structure (same node groups and dominant flows). Ensure that n=374 for ideological content and n=286 for financial content.

(E3): *Survivability / Kaplan–Meier (Figure 7) [10–20 human-minutes + 2–10 compute-minutes + ∼1GB disk; approximate]*
How to: Reproduce survivability curves and comparisons using `lifelines`.
Preparation: Ensure `kaplan_data.parquet` exists, are placed in the `data` folder, and Python dependencies are installed.
Execution: `python3 kaplan.py`
Results: The script should write one or more survivability plots into `./output/`.
Expected outcome: generated curves match the figure provided in the paper (Figure 7).

(E4): *DSL Modeling (supports Table 1 and related modeling outputs) [15–30 human-minutes + 10–30 compute-minutes + ∼1GB disk; approximate]*
How to: Run the R script that executes DSL experiments over the provided CSV inputs.
Preparation: Ensure `socialblade_v_baseline.csv` and `/fameswap_v_baseline.csv` exist, are placed in the `data` folder, and R can load `dsl`.
Execution: `Rscript run_R_exps.R`
Results: The script should print numeric results in the console output.
Expected outcome: the file exists and is non-empty; console output includes experiment summaries and variables with significant effects match those reported in Table 1).

A.5 Notes on Reusability

Beyond reproducing the analyses in this paper, the provided channel-level datasets can be reused to support longitudinal monitoring of repurposed channels. In particular, the channel identifiers and handle-change metadata can be used to (i) check whether the channels in our repurposed samples have been repurposed again since the observation window of this study, (ii) assess whether their content or topical focus has shifted over time, and (iii) track additional platform outcomes such as channel suspension/termination status and monetization status (where observable via public signals or permitted interfaces).

More broadly, the artifact includes both a set of repurposed channels and a randomly selected baseline sample of non-repurposed channels (subject to the same subscriber threshold). This enables future work that compares downstream outcomes for repurposed channels against "typical" channels, including extended follow-up windows, alternative outcome definitions, and additional covariates derived from subsequent observations.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Not All Those Who Share Are Lost: Analyzing 25 Years of Cybersecurity Artifact Sharing Practices Through Automated Discovery

Daan Vansteenhuyse[1], Arthur Bols[1], Lieven Desmet[1], Victor Le Pochat[2], Jo Van Bulck[1], Marton Bognar[1]

[1]DistriNet, KU Leuven
[2]Independent

A Artifact Appendix

A.1 Abstract

This artifact consists of three parts: (1) the ArtiFinder tool, capable of identifying artifact URLs in paper PDFs; (2) the output of ArtiFinder on 25 years of top-tier cybersecurity papers together with metadata such as authors and citation counts; and (3) scripts to reproduce our analysis and generate the figures from our paper. All scripts can be executed on a commodity UNIX-based machine with internet access.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

ArtiFinder identifies artifact URLs in published paper PDFs. Due to licensing constraints, our artifact does not contain the PDFs used to generate the full dataset. The metadata contains a link to all papers, allowing users to download the PDFs for subsequent experimentation with ArtiFinder. Our scripts run without modifying local data.

A.2.2 How to access

The initial version of ArtiFinder and our collected dataset is archived at https://doi.org/10.5281/zenodo.20412201. This deposit also includes all scripts necessary to perform our analysis.

We are continuing the development of ArtiFinder at https://github.com/DistriNet/ArtiFinder, where we invite contributions. We also maintain an active version of the dataset at https://github.com/DistriNet/ArtiFinder-Data, open to corrections and extensions.

A.2.3 Hardware dependencies

None.

A.2.4 Software dependencies

Our artifact can be executed on any UNIX-based system with Python 3.12 and Poppler[1].

A.2.5 Benchmarks

The artifact does not include the analyzed PDF files, but our raw data files in `data/` contain all metadata used in the analysis, together with links to the papers. Our tool ArtiFinder is designed to work with any paper PDF.

A.3 Set-up

A.3.1 Installation

We recommend using a Debian-based system, but any UNIX-based system should work, provided it has the software installed mentioned in Appendix A.2.4. Download the files from Zenodo or pull our GitHub repository and (optionally) set up a virtual Python environment.

For example, on Debian 12, the installation can be performed using the following commands:

```
sudo apt-get install python3-gi \
python3-gi-cairo gir1.2-gtk-4.0 libxml2 cmake \
pkg-config libcairo2-dev girepository-2.0 \
libpoppler-glib-dev gir1.2-poppler-0.18
python3 -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Some heuristics use the GitHub API to look up metadata, it is therefore recommended to create a personal access token and add it to `.env` as shown in `.env.example`. This token can be created for free at https://github.com/settings/personal-access-tokens.

[1] https://poppler.freedesktop.org/

A.3.2 Basic Test

`python test_install.py` checks if all the required packages are installed and should not return any output.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): ArtiFinder is capable of extracting links from a paper PDF and ranking them to identify the most likely artifact link. This is shown by experiment E1 and is described in Section 3.

(C2): ArtiFinder is able to identify artifacts links in cybersecurity paper PDFs with a high (93.3–98.8%) accuracy compared to three different ground truth datasets. By conducting an ablation study, we show that multiple heuristics are needed to achieve this accuracy. This is shown by experiment E2 and is described in Section 3.3.

(C3): By running ArtiFinder on a corpus of 9,054 papers, we identify trends and statistics in artifact sharing across top-tier cybersecurity literature. All statistics and figures are generated in experiment E3 and are described in Section 4.

A.4.2 Experiments

(E1): [5 human-minutes + 2 compute-minutes + 10MB disk]:
How to: Due to licensing constraints, we cannot release paper PDFs. However, ArtiFinder works on any paper provided in PDF format, which the user can provide. Our raw data (`data/data.json`) contains links to all the papers analysed in our research.
Preparation: Ensure all Python dependencies are installed and the paper PDF is downloaded. The necessary metadata can be inputted directly into the CLI or be formatted as a JSON file containing the fields: title, authors, year and conference.
Execution:
```
python artifinder-cli.py \
--pdf /path/to/paper.pdf \
--json /path/to/metadata.json \
-o /path/to/output.json
```
Optionally, the metadata can be added using `--authors`, `--year`, `--conf`, `--title`.
Results: The output is written to the supplied output file and contains the metadata extended with the extracted links and the discovered artifact(s).

(E2): [1 human-minute + 5 compute-minutes + 200MB disk]:
How to: By comparing the raw ArtiFinder output with groundtruth datasets, the accuracy numbers can be recalculated.
Preparation: Install all Python dependencies.
Execution:
```
python reproduce_results.py \
        --experiment E2
```
Results: The output can be found in `results.md`, which should match the numbers in the paper and `expected_results.md`.

(E3): [1 human-minute + 5 compute-minutes + 200MB disk]:
How to: Run the analysis scripts to reproduce the various claims made about our dataset.
Preparation: Install all Python dependencies.
Execution:
```
python reproduce_results.py \
        --experiment E3
```
Results: The output can be found in `results.md` which should match the numbers in the paper and `expected_results.md`. The output is formatted per paper (sub)section. Generated figures are stored in `/figs`.

A.5 Notes on Reusability

ArtiFinder is designed to be modular, allowing adaptations with more heuristics depending on the use-case. On our GitHub, we invite contributions to optimize our artifact discovery process, possibly by fine-tuning it for other computer science fields. We hope ArtiFinder allows researchers to better map artifact availability trends to promote open and reproducible science. At https://github.com/DistriNet/ArtiFinder-Data, we release our dataset, open for corrections of mistakes made by ArtiFinder.

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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: PatchWeaver

Rui Li
Dawn Computing
mark.li@dawncomputing.com

Shuang Cao
Dawn Computing
scao@dawncomputing.com

A Artifact Appendix

A.1 Abstract

This artifact contains the PatchWeaver control-plane implementation, the ChangeSpec policy interface, deterministic remediation workloads, adversarial stress tests, ablation and sensitivity evaluators, overhead micro-benchmarks, raw result files, and figure-generation scripts. The artifact has received the *Artifacts Available*, *Artifacts Functional*, and *Results Reproduced* badges. The artifact exercises the real PatchWeaver state graph, ChangeSpec policy engine, rollout scorer, action vocabulary, and transition model in a deterministic discrete-event evaluation. The evaluation reruns the experiments, regenerates result JSON files and figures, and validates the paper's main claims: trajectory-level planning reduces policy violations, multi-step attack scenarios are blocked or escalated, ablations identify the key contributors, and policy-evaluation overhead remains practical.

A.2 Description & Requirements

The archive includes Rust crates for PatchWeaver, ChangeSpec, benchmarks, and evaluation utilities; workload and attack specifications; default policies and configuration; generated JSON results; paper-result figures; and scripts for building, running, and plotting the artifact. The artifact supports the paper's claims about policy-bounded remediation by executing the same policy and trajectory-scoring logic on controlled workloads.

A.2.1 Security, privacy, and ethical concerns

The artifact uses synthetic Kubernetes-style workloads and simulated attack scenarios. By default it does not connect to a live Kubernetes cluster, cloud account, ticketing system, registry, LLM provider, or evaluator data source. The attack experiments are simulations that write result files under the artifact directory; they do not launch privileged containers, modify host security settings, or perform destructive operations. No private user data is included.

A.2.2 How to access

The permanent artifact archive is available from Zenodo through the concept DOI `https://doi.org/10.5281/zenodo.20477169`, which resolves to the latest published version while retaining access to earlier versions. Download the archive, unpack it, and enter the artifact directory containing `Cargo.toml` and `scripts/`.

A.2.3 Hardware dependencies

None. A standard CPU-based environment is sufficient. We recommend at least 4 CPU cores, 8 GB RAM, and 10 GB free disk space for Rust build artifacts and generated results. No GPU, trusted execution environment, FPGA, custom hardware, Kubernetes cluster, or GPT/API credential is required for the Phase-2 artifact evaluation.

A.2.4 Software dependencies

The artifact has been prepared for common Unix-like environments such as Ubuntu 22.04 LTS or macOS. Required software is Rust 1.75 or newer with Cargo and Python 3.8 or newer with `venv` support. The helper script `scripts/setup.sh` creates an isolated `.venv`, installs the versions of `numpy` and `matplotlib` declared in `requirements.txt`, and builds the benchmark runner. It does not use Ubuntu's system `python3-matplotlib` package.

A.2.5 Benchmarks

The benchmark suite contains four remediation workloads: ImageRebuild, RBAC Minimization, Canary Gating, and PolicyDrift Recovery. It also contains ten Kubernetes security stress-test categories, an ablation study, horizon and penalty

sensitivity sweeps, and a policy-evaluation latency benchmark.

A.3 Set-up

A.3.1 Installation

After downloading and unpacking the Zenodo archive:

```
cd <artifact-directory>
bash scripts/setup.sh
```

The setup script builds the main PatchWeaver binary and the benchmark runner. If Rust is already installed, the core build step can also be run directly:

```
cargo build --release -p benchmarks
```

A.3.2 Basic Test

Run a small workload to verify that the evaluator, policy engine, rollout scorer, and result writer are working:

```
mkdir -p /tmp/patchweaver-smoke
./target/release/benchmarks evaluate \
  --workload imagerebuild \
  --num-episodes 2 \
  --seed 42 \
  --output /tmp/patchweaver-smoke \
  --baselines
ls /tmp/patchweaver-smoke
```

Expected outcome: the command completes without errors and writes JSON result files in `/tmp/patchweaver-smoke`. Results are deterministic for the specified seed.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): PatchWeaver's artifact includes the main implementation components needed to exercise policy-bounded remediation: typed state graph, ChangeSpec policy predicates, plan proposer interface, rollout scorer, executor stubs, and audit logging. This is checked by the build and basic test.

(C2): In the artifact evaluation, trajectory-level planning (PatchWeaver and Template+Rollout) produces lower policy-violation rates than myopic, partial-policy, and no-policy baselines. This reproduces the main Section 7 claim that trajectory-level compliance checking is the primary driver of safer autonomous remediation.

(C3): The security stress tests show that explicit policies plus trajectory scoring block or escalate multi-step adversarial scenarios that evade point-in-time checks.

(C4): The ablation, sensitivity, and overhead experiments reproduce the paper's component-level claims: removing rollout simulation or drift detection substantially increases violations, while policy-evaluation latency remains practical for admission-style use.

A.4.2 Experiments

(E1): Main workload evaluation [5 human-minutes + 10–30 compute-minutes]:
Preparation: Build the benchmark runner using the Installation command above.
Execution: Run:
`export SEED=42`
`bash scripts/run_experiments.sh`
Results: Inspect `baseline_comparison.json` under `results/workloads/`. PatchWeaver and Template+Rollout should remain among the lowest-violation methods, reproducing the paper's main comparative claim that trajectory-level planning is safer than myopic, partial-policy, and no-policy baselines.

(E2): Security stress tests [included in E1, or 5–10 compute-minutes when run separately]:

```
./target/release/benchmarks attack \
  --category all \
  --num-episodes 30 \
  --seed 42 \
  --output results/attacks
```

Expected output is `attack_results.json` under `results/attacks/`, summarizing block, escalation, miss, and dwell-time behavior for ten attack categories. The output shows that multi-step attack scenarios are rejected or escalated at high rates, reproducing the paper's attack-resistance claim.

(E3): Ablation, sensitivity, overhead, and figures [included in E1]:

```
./target/release/benchmarks ablation \
  --num-episodes 50 \
  --seed 42 \
  --output results/ablation
./target/release/benchmarks overhead \
  --num-predicates 32 \
  --iterations 1000 \
  --output results/overhead
./.venv/bin/python \
  scripts/generate_figures.py \
  --input results/ \
  --output figures/
```

Expected output includes JSON summaries under `results/` and regenerated PDF figures under `figures/`. The ablation output should show large violation increases when rollout simulation or drift detection is removed, and the overhead output should show practical policy-evaluation latency.

A.4.3 Output-to-paper mapping and scope

The archive's `AE_OUTPUT_MAPPING.md` provides the complete mapping. In summary, `baseline_comparison.json` supports Table 2 and Figure 3; the four workload JSON files support Figure 4; `attack_results.json` supports Table 3 and the standalone artifact-mode Figure 5; `ablation_results.json` supports Table 4; and `overhead_results.json` supports the artifact-mode Figure 6.

The standalone Figure 5 reports PatchWeaver attack dwell times and aggregate outcomes. Baseline dwell-time streams from the original cluster evaluation are not included in the standalone JSON and are therefore not regenerated. The Figure 6 artifact measures local in-process policy-engine latency; webhook round-trip and cluster CPU/memory values require the original cluster and are not reported as standalone measurements.

A.5 Notes on Reusability

Researchers can add workloads under `data/workloads/`, attack scenarios under `data/attacks/`, and policy rules under `data/policies/default_policy.dsl`. The benchmark crate exposes the workload, baseline, ablation, sensitivity, and overhead drivers used by the submitted 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/usenixsec2026/.



USENIX Security '26 Artifact Appendix: Trustworthy and Confidential SBOM Exchange

Eman Abu Ishgair[†] Chinenye Okafor[†] Marcela S. Melara[‡] Santiago Torres-Arias[†]

[†]Purdue University, {eabuishg, okafor1, santiagotorres}@purdue.edu

[‡]Intel Corporation, marcela.melara@intel.com

A Artifact Appendix

A.1 Abstract

This artifact appendix provides a reference implementation of Petra, a system for confidentiality- and integrity-preserving Software Bill of Materials (SBOM) exchange. Petra enables selective disclosure of SBOM fields using Ciphertext-Policy Attribute-Based Encryption (CP-ABE) while preserving verifiable integrity through Merkle-authenticated SBOM trees. The artifact includes: (i) Petra's implementation, (ii) CP-ABE bindings, (iii) SBOM preprocessing and evaluation scripts to reproduce the performance and storage overhead results reported in the paper. Evaluators can reproduce all major experimental claims either by installing Petra directly on their machine or using the containerized setup.

A.2 Description & Requirements

The artifact evaluates Petra's ability to securely encrypt, decrypt, and verify redacted SBOMs at scale. Experiments measure: tree construction cost, Merkle hashing cost, encryption and decryption overhead, storage overhead across plaintext, encrypted, and decrypted SBOM trees.

The evaluation uses Chainguard's bom-shelter SBOM dataset containing SPDX and CycloneDX SBOMs of varying sizes and complexity.

A.2.1 Security, privacy, and ethical concerns

The artifact does not require disabling security mechanisms, elevated privileges (except installing dependencies), or access to sensitive personal data. The evaluation processes publicly available SBOM datasets and locally generated cryptographic keys. No network services are contacted except for dependency installation and dataset retrieval. The included Key Management Service (KMS) runs locally for testing purposes only.

A.2.2 How to access

The artifact is available as a public, git repository:

```
git clone https://github.com/TSELab/SBOMCtl.git
cd SBOMCtl
git checkout \
d49dbf6c82202dd40eb5cb68b8eb375741495fcc
```

We also made a git-archived version available via Zenodo: https://doi.org/10.5281/zenodo.17906622

The repository contains:

1. Petra source code
2. Evaluation scripts
3. Configuration files
4. Dockerfile for containerized evaluation

A Docker image containing all required dependencies for Petra and its evaluation pipeline, can also be rebuilt locally using the Dockerfile included in the source archive.

A.2.3 Hardware dependencies

None. The evaluation was carried out on a commodity x86-64 machine equipped with a 2.10 GHz Intel Xeon E5620 CPU and 256 GB of RAM. No specialized hardware (e.g., GPUs, FPGAs, or hardware accelerators) is required to evaluate this artifact.

A.2.4 Software dependencies

Local machine setup Linux or macOS, Python >= 3.9, Rust toolchain (rustc, cargo), OpenSSL, Docker.

Python dependencies Installed via pip install -r requirements.txt

Rust dependencies Built via maturin for CP-ABE bindings

A.2.5 Benchmarks

The evaluation uses the Chainguard's bom-shelter SBOM corpus, which contains SPDX and CycloneDX SBOMs between 1 and 5000 packages per SBOM. The dataset access methods:

1. Direct clone from GitHub:

```
# From artifact root directory(SBOMCtl)
cd sbom-data
git clone \
https://github.com/chainguard-dev/bom-shelter
cd bom-shelter
git checkout \
89ed4943e027c9bf148e6a80d7b2b78ae8d7771a
```

2. Zenodo: https://doi.org/10.5281/zenodo.17859760 Download into `SBOMCtl/sbom_data` directory

A.3 Set-up

A.3.1 Installation

Option A: Docker (recommended) For a fully containerized setup (that has all the petra setup and dataset), you can run Petra using the provided Docker file. This option packages Petra, cpabe bindings, and the bom-shelter dataset inside the container, so you can start directly from preprocessing and evaluation.

The following commands build and start the evaluation container:

```
# From artifact root directory(SBOMCtl)
docker build -f dockerfiles/petra_evaluation \
-t petra-eval .

docker run -d --name petra-eval petra-eval \
tail -f /dev/null
```

To enter the container at anytime:

```
docker exec -it petra-eval bash
```

Option B: Native (non-containerized) Setup You can set up a virtual environment in two ways:

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
make init
```

or

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
virtualenv venv
source venv/bin/activate
pip install -r requirements.txt
```

Build CP-ABE bindings cpabe is the python bindings for the rust crate rabe, this is needed to support CP-ABE functionality. Before you run any other code, make sure that these bindings are built by issuing:

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
sudo apt install rustc cargo
cd src/cpabe
maturin develop
```

This will build the rust code, as well as install an editable version of the python bindings into the current virtual environment.

Install Petra Petra is the main system. If cpabe is installed, you should be able to install the petra package:

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
pip install -e .
```

SBOM Signing Test Key Generation generates the SBOM signing and verification keys for Petra. Assuming you have openssl installed, generate DER format ECDSA keys:

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
cd tests
openssl ecparam -name prime256v1 -outform der\
-genkey -out privkey.der -noout
openssl ec -inform der -in privkey.der -pubout\
-outform der -out pubkey.der
```

Start the KMS Service The KMS service handles the key management, to start it run:

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
python src/petra/internals/kms.py
```

A.3.2 Basic Test

After the setup, you should be able to run the Petra CLI by running the test files under /tests. To run a functional encryption and decryption test including functional KMS:

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
python tests/test_models.py
```

Expected outcome:

```
Printing raw SBOM tree
done encrypting
Printing hashed and encrypted SBOM tree
Merkle Root Hash for SBOM:
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): Petra enables selective disclosure of SBOM fields while preserving integrity across redistribution. Demonstrated by encryption, decryption, and membership-proofs.

(C2): Petra introduces <13% storage overhead for encrypted SBOMs on average. Demonstrated by storage measurements in Section 6.2.1 and reproduced via evaluation scripts.

(C3): CP-ABE encryption and decryption overhead scales with SBOM size and node count but remains practical. Demonstrated by timing measurements in Figures 5 and 6.

A.4.2 Experiments

The following steps run Petra against two policies: security weaknesses, and intellectual property policy, which are two separate experiments that we run at once.

Preprocessing [20–30 compute-minutes]

Execution: This step parses SBOMs, tries to build SBOM trees and filters SBOMs with erroneous formats.

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
python evaluation/preprocess_sboms.py
```

Results: Valid SBOMs copied to `input_sboms` directory

Performance Evaluation [8 - 10 compute-hours]:

Execution: This step executes the following script to build the SBOM tree, encrypt and decrypt SBOM trees, verify integrity properties and record timing results (C1).

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
python evaluation/test_evaluations_AES.py
```

Results: For each tested policy, Petra saves the performance results in a JSON file named after the policy under `privateSBOMExchange/output` directory, the results contains the following for each SBOM file: file size, build tree time, hash time, encrypt time, decrypt time, sbom tree storage, encrypted tree storage, decrypted tree storage, tree nodes count, policy

Plot Generation [5 minutes]:

Execution: The generated performance files in Performance Evaluation are automatically consumed by the plotting scripts in evaluation/, which aggregate metrics across policies and produce comparative performance visualizations.

Generate performance plots with:

```
# From artifact root directory(SBOMCtl)
cd privateSBOMExchange
python evaluation/plot_perf_size_AES.py
```

Results: For each tested policy, the corresponding plots are automatically saved in a subdirectory named after the policy inside the `privateSBOMExchange/output` directory. This step should reproduce encrypt time vs tree nodes count plots that correspond to figure 5. It also reproduces decrypt time vs tree nodes count plots that correspond to figure, these plots cover C3.The script finally calculates the following statistics:

1. Average Percentage Increase from SBOM Tree Size to Encrypted Tree Size.
2. Average Percentage Increase from Encrypted Tree Size to Decrypted Tree Size.
3. Ratio of encrypted tree storage to SBOM tree storage.
4. Mean encryption – decryption time difference.
5. Mean decryption time percentage.

The computed statistics average percentage increase from SBOM tree size to encrypted tree size, and average percentage increase from encrypted tree size to decrypted tree size corresponds to encryption and decryption storage overhead results in Section 6.2.1 of the paper (C2).

The computed statistics mean decryption time percentage corresponds to decryption time percentage in the Abstract and the Introduction.

By comparing your generated plots and statistics with those in the paper, you can verify that Petra exhibits the expected performance behavior on your dataset or configuration.

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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Concurrency Fuzzing of the Linux Kernel with eBPF

Jiacheng Xu*
Zhejiang University†

Dylan Wolff*‡
National University of Singapore

Xing Yi Han
National University of Singapore

Jialin Li
National University of Singapore

Abhik Roychoudhury
National University of Singapore

Abstract

Concurrency is indispensable for modern software systems to meet performance and scalability demands, yet concurrency bugs remain notoriously difficult to detect and reproduce. Controlled Concurrency Testing (CCT) mitigates this challenge by systematically exploring thread interleavings through scheduling control. However, existing CCT approaches for OS kernels largely rely on external enforcement mechanisms, such as custom hypervisors or invasive kernel patches, resulting in substantial overhead and limited maintainability and extensibility. In this work, we present SECT, the first kernel-native concurrency fuzzing framework that rethinks scheduling as a first-class exploration mechanism. SECT introduces a novel CCT scheduler with temporal isolation scheduling and embeds programmable scheduling policies directly into the kernel dispatch path via eBPF, enabling fine-grained control over thread interleavings without customized hypervisors or extensive kernel core modification. In addition, SECT provides a preemption-safe instrumentation mechanism for injecting scheduling points at critical kernel events and incorporates a two-phase fuzzing workflow to jointly explore both sequential and concurrent behaviors. Our evaluation demonstrates that SECT achieves 38% more branches, 57% overhead reduction and 11.4× speed-up in bug exposure compared to a leading state-of-the-art kernel concurrency fuzzer. Moreover, SECT discovers eight previously unknown concurrency-related bugs in the Linux kernel, six of which have already been confirmed and fixed by developers.

A Artifact Appendix

A.1 Abstract

This artifact contains SECT, as well as infrastructure for setting it up on a recent target kernel (v6.13-rc4), our benchmark of kernel concurrency bugs used in our evaluation, and data from our real-world bug-finding runs.

A.2 Description & Requirements

Generally, the system requirements for this artifact are (1) a relatively recent version of Ubuntu (tested on 24.04) (2) Docker (3) QEMU with KVM enabled. In terms of hardware recommend a machine with at least 8 logical CPU cores and 16GB of RAM. All other setup should be handled by scripts in the SECT repository.

A.2.1 Security, privacy, and ethical concerns

The risks of using SECT are minimal, especially if used within a VM, as is the documented setup in our artifact. If you attempt to run SECT on your host system outside of a VM, any resulting kernel panics will of course affect your host system directly.

A.2.2 How to access

Our artifact is available open-source on Github at `https://github.com/m0ck1ng/sect` and also permanently archived at `https://doi.org/10.6084/m9.figshare.32468280`

A.2.3 Hardware dependencies

SECT does not require any specific hardware, though we recommend a multi-core system with at least 8 logical cores for performance when running fuzzing campaigns and to ensure that the Linux kernel itself can be rebuilt by your system in a reasonable amount of time. To reproduce the full fuzzing experiments, we recommend a machine with at least 80 cores and 2GB of RAM per core, to allow all 10 trials to be run in parallel.

*Both authors contributed equally
†Research conducted primarily at the National University of Singapore
‡Corresponding Author: `wolffdy0@gmail.com`

A.2.4 Software dependencies

We have developed and tested SECT on Ubuntu 24.04. Additional recent versions of Ubuntu or Debian are also likely to work without modification. Other Linux distributions may also work, but are untested and will require some slight modifications of shell scripts (e.g. to install dependencies from `rpm` instead of `apt`).

Additionally, the host system needs Docker installed to use our provided setup. Docker is used purely for a hermetic build environment to ensure all other dependencies are the correct versions. For example, SECT and the kernel under test are mounted into a docker container as a volume, and built inside the container rather than on the host system.

The fuzzing setup uses a forked version of syzkaller, which runs in one or more virtual machines using QEMU. QEMU is installed as part of the build script automatically on the host system via `apt`.

Finally, to create a disk image for the guest VM, `debootstrap` is also a required dependency, installed automatically by the main build script.

A.2.5 Benchmarks

We have provided the details of the ten concurrency bugs in the benchmark from our experiments in our artifact, including reproducing inputs and a kernel patch for v6.13-rc4 which introduces all of the bugs into that kernel version.

A.3 Set-up

All that is needed for setup is to clone the SECT repository:

```
git clone https://github.com/m0ck1ng/sect.git
```

And to install Docker:

https://docs.docker.com/engine/install/

A.3.1 Installation

We have consolidated the installation into a single script:

```
./build.sh
```

Note that some operations may require superuser permissions to e.g. install dependencies or mount a volume for disk image creation.

A.3.2 Basic Test

The best way to check that SECT is working after installation is to start a fuzzing campaign via:

```
./syzkaller/bin/syz-manager
-config configs/syzkaller.cfg.example
```

The configuration should have been pre-populated by the installation process, but it can be edited according to your system requirements as necessary. You should see output at the command line and in the web interface (`http://0.0.0.0:56741`) containing:

```
Using thread scheduler: <path>
```

and later you should see that `schedCollided` increases over time, e.g.:

```
VMs 4, executed 2434, schedCollided 650, cover
29526, signal 38453/47876, crashes 0, repro 0,
triageQLen 8817
```

A.4 Evaluation workflow

A.4.1 Major Claims

(C1): SECT achieves higher fuzzing throughput and coverage that other tools, shown in Figure 3 and Table 2.

(C2): SECT is more effective at finding concurrency bugs than the native scheduler under Syzkaller or SegFuzz, shown by Table 3 and Figure 4.

A.4.2 Experiments

(E1) This experiment reproduces the code coverage and throughput results for SECT (C1) as well as the Bug-Finding Results (C2) shown in Table 3. Please note that for all experiments to finish in 48 hours, all trials must be run simultaneously, requiring at least 80 CPU cores for SECT (4 VM instances * 2 cores per instance * 10 trials).

[6 Hours Human] [48 Hours Compute]

1. *Preparation.*Build the benchmark kernel with all 10 concurrency bugs by applying the patch provided in our artifact and building SECT with the provided script.
2. *Experiment.* Run a fuzzing campaign via `syz-manager` as outlined in the artifact using the provided configuration for each *trial*, using a timeout of 48 hours.
3. *Results.* For each crash found by SECT, manually check if it corresponds to any of the bugs in the benchmark. Note that once a bug has been correlated, any crashes from other trials with the same hash can also be mapped to that bug. We compute throughput and latency from the total number of executions reported by `syz-manager` and the 48-hour time period. To obtain coverage data we use the documented utility provided by Syzkaller: https://github.com/google/syzkaller/blob/master/docs/coverage.md

We note that for comparing with Syzkaller, most of the above steps are identical, save for using a clean build of Syzkaller in lieu of SECT. For SegFuzz, we use the instructions provided by the tool authors. For Snowboard, we similarly use the instructions provided by the tool authors.

(E2) This experiment reproduces the survival plots in Figure 4 (C2). Unfortunately, we didn't find a good way to automate these experiments, so they are currently interactive and thus take a very long time to reproduce.
[>10 Hours Human + Compute (interactive)]

1. *Preparation.* Build the benchmark kernel with all 10 concurrency bugs by applying the patch provided in our artifact and building SECT with the provided script.

2. *Experiment.* Start a VM instance using the script provided in our artifact `./run_qemu.sh`. Copy all necessary binaries and programs into the guest with the provided script `./copy.sh`. Use SSH to obtain a second terminal window inside the guest. In one window, start the sched-ext scheduler `./scx_serialise` with the desired scheduling algorithm via `-r`. In the other window, execute the desired reproducer with `syz-execprog` using two processes and 10000 repetitions.

3. *Results.* For each crash, we record the number of schedules until the crash as logged by the window with `scx_serialise`. We generate the survival plots using the script provided in our artifact (`scripts/analysis/plot.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/usenixsec2026/.



USENIX Security '26 Artifact Appendix
XGUARDIAN: Towards Generalized, Explainable and More Effective Server-side Anti-cheat in First-Person Shooter Games

Jiayi Zhang
The University of Hong Kong
brucejiayi@connect.hku.hk

Chenxin Sun
The University of Hong Kong
roniny@connect.hku.hk

Chenxiong Qian
The University of Hong Kong
cqian@cs.hku.hk

A Artifact Appendix

This is the artifact evaluation document for XGUARDIAN, which is going to appear at USENIX Security '26.

A.1 Abstract

You may follow the instructions below to access, reuse, and reproduce XGUARDIAN as well as the experiments mentioned in our paper.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Data Source and Acquisition. All data used in this study were provided by a third-party competitive gaming platform *5EPlay* that hosts CS2 servers independently of *Valve Corporation* (the game's developer). These servers operate under the platform's own user agreements and privacy policies, while authentication occurs via Steam. *5EPlay* directly provided the de-identified dataset to the research team, in compliance with their platform policies[1]. No private information or personally identifiable data was disclosed to the authors. Players' re-identifiable information was removed prior to data transfer.

Data Usage Agreement. The dataset is governed by the *5EPlay Personal Information Protection Policy*, which permits sharing de-identified information for academic research purposes conducted in the public interest. While the data usage agreement acknowledged by players was not specific to this research, it explicitly allows the platform to share anonymized gameplay data for research. All data were handled with strict confidentiality and in accordance with applicable legal and ethical standards.

EXPLAINER Potential Impact. EXPLAINER could theoretically influence reviewers' decisions, potentially increasing the risk of false decisions. To mitigate this risk, EXPLAINER is strictly a supportive interpretability tool intended to help experienced game masters (GMs) quickly identify anomalous gameplay segments. All decisions are made independently by trained GMs, and bans are only enforced when multiple reviewers reach the same conclusion, minimizing the influence of any single reviewer or algorithm. GMs should be trained to know that XGUARDIAN can make mistakes. Players affected by false positives retain the right to appeal, triggering review by a separate moderation team.

IRB Review. This research was reviewed and approved by the Human Research Ethics Committee (Reference Number: EA250702) at the author's affiliated institution. The study design, data acquisition, and handling procedures were deemed compliant with human subjects research standards.

[1] *5EPlay Personal Information Protection Policy*, translated by the authors, original in Simplified Chinese: https://arena.5eplay.com/page/privacy.

A.2.2 How to access

We open-source XGUARDIAN's repository, datasets, and artifact at https://doi.org/10.5281/zenodo.17845613 (concept DOI, link to the latest version of the artifact), as well as its raw CS2 data listed in Table 1.

Table 1: XGUARDIAN's raw data.

Part	URL
1	https://doi.org/10.5281/zenodo.17838584
2	https://doi.org/10.5281/zenodo.17838763
3	https://doi.org/10.5281/zenodo.17838827
4	https://doi.org/10.5281/zenodo.17838865
5	https://doi.org/10.5281/zenodo.17839153
6	https://doi.org/10.5281/zenodo.17844922
7	https://doi.org/10.5281/zenodo.17844928
8	https://doi.org/10.5281/zenodo.17844938
9	https://doi.org/10.5281/zenodo.17844942
10	https://doi.org/10.5281/zenodo.17844950
11	https://doi.org/10.5281/zenodo.17844964
12	https://doi.org/10.5281/zenodo.17844969
13	https://doi.org/10.5281/zenodo.17844977

A.2.3 Hardware dependencies

Technically, any PC with CPU is able to reuse XGUARDIAN directly, and any PC with NVIDIA GPU(s) can train and reproduce XGUARDIAN. In terms of the experimental configurations, XGUARDIAN is developed and tested on two different servers. XGUARDIAN's preprocessing is implemented on an Ubuntu server equipped with 128 cores AMD EPYC 7713P 64-Core Processor and 256 GB RAM. The rest of the XGUARDIAN's experiments including training, validation, testing, metric calculations, and other server-side experiments are conducted on an Ubuntu server equipped with a 24-core Intel Xeon Gold 5418Y CPUs, 503 GB RAM and eight NVIDIA GeForce RTX 4090 24GB (only used two of them for the training).

A.2.4 Software dependencies

XGUARDIAN uses go version `go1.22.5 linux/amd64` for preprocessing. The Go compiler can be downloaded from https://go.dev/dl/. Before parsing, use the following command to ensure that `go` is available in your environment.

```
$ export PATH=$PATH:/usr/local/go/bin
```

XGUARDIAN uses NVIDIA-SMI `570.124.06`, driver version `570.124.06`, CUDA `12.8`, TensorFlow `2.17.0`, and cuDNN `8` for training XGUARDIAN. Similar to all ML-based solutions, these are the basic prerequisites for an ML-based project. This project uses `conda` to manage packages and environments. This project is implemented in Python (https://www.python.org/downloads/) version `3.11.8`.

A.2.5 Benchmarks

Three datasets are required: CS2 (main dataset used for most of the experiments), FARLIGHT84 and HAWK (two sets are used for the comparision and generalizability experiments). All datasets are included in Appendix A.2.2.

A.3 Evaluation workflow

The evaluation can be proceed in the following sequence.

Step 1. Preprocessing. This step is optional if you decide to use our processed datasets and/or results. For the details, please refer to Appendix A.5.

Step 2. Deploy INSPECTOR. It is the implementation of XGUARDIAN's detection function. For the detailed usage, please refer to Appendix A.6.1.

Step 3. Deploy EXPLAINER. It is the implementation of XGUARDIAN's interpretability. For the detailed usage, please refer to Appendix A.6.2.

Step 4. Reproduce the experiments. Please refer to Appendix A.7.

A.3.1 Major Claims

(C1): This artifact is able to be publicly available and permanently accessed.

(C2): This artifact is able to reuse XGUARDIAN, its datasets, trained models, codes and results.

(C3): This artifact is able to reproduce our paper experimental results.

A.4 Set-up

[*Installation*] Create a Conda environment for preprocessing. Replace the `prefix` field in `preprocessing_clean.yml` with your local `conda` path.

```
$ conda env create -f XGuardian/XGuardian/env/
    ↪ preprocessing_env/preprocessing_clean.yml
$ conda activate XGuardian-preprocessing
$ pip install -r XGuardian/XGuardian/env/
    ↪ preprocessing_env/
    ↪ preprocessing_pip_requirements.txt
```

Create a detector environment. Replace the `prefix` field in `detector_clean.yml` with your local `conda` path.

```
$ conda env create -f XGuardian/XGuardian/env/
    ↪ detector/detector_clean.yml
$ conda activate detector_clean
$ pip install -r XGuardian/XGuardian/env/detector/
    ↪ detector_pip_requirements.txt
```

Create an explainer environment. Replace the `prefix` field in `explainer_clean.yml` with your local `conda` path.

```
$ conda env create -f XGuardian/XGuardian/env/
    ↪ explainer/explainer_clean.yml
$ conda activate explainer_clean
$ pip install -r XGuardian/XGuardian/env/explainer
    ↪ /explainer_pip_requirements.txt
```

[*Troubleshooting*] If you are using an archived version of the artifact (v1), you might encounter the following issue. We use both `conda install` and `pip install`. When exporting the environment, `pip freeze` might capture the absolute path of the packages instead of their version. Might encounter an exception below.

```
ERROR: Could not install packages due to an
    ↪ OSError: [Errno 2] No such file or
    ↪ directory: /croot/xxx
```

To solve this problem, when a line is inside `requirement.txt` that contains `@ file:///croot`, discard everything starting from `@`. For example, below. Alternatively, you may download our revised version of the affected environments from: https://doi.org/10.5281/zenodo.18194986. This issue is fixed in the latest version of our artifact.

```
Bottleneck @file:///croot/bottleneck_1707864210935
    ↪ /work --> Bottleneck
```

A.5 Preprocessing

The raw data can be found at Zendo (please check Table 1 above). The raw data is **EXTREMELY** large, we strongly recommend you to use our processed data (can be found in `XGuardian/XGuardian/Design/dataset`). **If you would like to use our processed dataset directly, please skip this entire subsection**.

A.5.1 Parsing

Download the raw data and unzip the files.

```
$ cat XGuardian_raw.tar.zst.part.* > XGuardian_raw
    ↪ .tar.zst
$ tar -I zstd -xf XGuardian_raw.tar.zst
```

The initial file structure should be prepared as follows.

```
.
`-- demo_dir/
    |-- demo_0.zip
    |-- demo_1.zip
    |-- ...
    `-- demo_x.zip
```

We use the absolute path when zipping the artifact. You should rename the path and delete `home2/data1/J.Zhang/` ahead of the `XGuardian/XGuardian/`. Activate the conda environment `XGuradian-preprocessing`. To process concurrently, partition the demos first. You need to enter the demos' root directory `./demo_dir` in the command line.

```
$ python partition_creation.py
```

The file structure will then be partitioned as follows.

```
.
`-- demo_dir/
    |-- partition_0/
    |   |-- demo_0.zip
    |   |-- demo_1.zip
    |   |-- ...
    |   `-- demo_x.zip
    |-- ...
    `-- partition_y/
        |-- demo_x+n.zip
        |-- demo_x+n+1.zip
        |-- ...
        `-- demo_x+n+m.zip
```

To parse the demos concurrently, simply run `shell` script as follows. Note that you should manually modify the number of partitions in the `for i in {0..y}` loop, where `y` represents the total number of partitions, the `SCRIPT_PATH` where you store the source codes, and the `BASE_DIR` where it is the demos' root directory.

```
$ chmod +x run_preprocessing_abwh.sh
$ ./run_preprocessing_abwh.sh
```

If processing considerable demos, it will take a long time. After the parsing, the file structure should look like this. The `log.txt` under each partition folder contains error messages while parsing the demos.

```
.
`-- demo_dir/
    |-- partition_0/
    |   |-- demo_0/
    |   |   |-- demo_0_bomb.csv
    |   |   |-- demo_0_damages.csv
    |   |   |-- demo_0_header.txt
    |   |   |-- demo_0_infernos.csv
    |   |   |-- demo_0_kills.csv
    |   |   |-- demo_0_smokes.csv
    |   |   |-- demo_0_weapon_fires.csv
    |   |   |-- demo_0.dem
    |   |   `-- demo_0.zip
    |   |-- ...
    |   |-- demo_x/
    |   |   |-- demo_x_bomb.csv
    |   |   |-- demo_x_damages.csv
    |   |   |-- demo_x_header.txt
    |   |   |-- demo_x_infernos.csv
    |   |   |-- demo_x_kills.csv
    |   |   |-- demo_x_smokes.csv
    |   |   |-- demo_x_weapon_fires.csv
    |   |   |-- demo_x.dem
    |   |   `-- demo_x.zip
    |   `-- log.txt
    |-- ...
    `-- partition_y/
        |-- demo_x+n/
        |   |-- demo_x+n_bomb.csv
        |   |-- demo_x+n_damages.csv
        |   |-- demo_x+n_header.txt
        |   |-- demo_x+n_infernos.csv
        |   |-- demo_x+n_kills.csv
        |   |-- demo_x+n_smokes.csv
        |   |-- demo_x+n_weapon_fires.csv
        |   |-- demo_x+n.dem
        |   `-- demo_x+n.zip
        |-- ...
        |-- demo_x+n+m/
        |   |-- demo_x+n+m_bomb.csv
        |   |-- demo_x+n+m_damages.csv
        |   |-- demo_x+n+m_header.txt
        |   |-- demo_x+n+m_infernos.csv
        |   |-- demo_x+n+m_kills.csv
        |   |-- demo_x+n+m_smokes.csv
        |   |-- demo_x+n+m_weapon_fires.csv
        |   |-- demo_x+n+m.dem
        |   `-- demo_x+n+m.zip
        `-- log.txt
```

Run the following script to check the integrity and filter the redundant data. The redundant data will be moved to a new directory under the root directory named `redundant`. If exceptions occurred, detailed logs will be presented in `integrity_check_log.txt`.

```
$ python after_parse_itegrity_check.py <
    ↪ DEMO_ROOT_DIR>
```

The file structure should then look like this.

```
.
`-- demo_dir/
    |-- partition_0/
    |   |-- demo_0/
    |   |   |-- demo_0_bomb.csv
    |   |   |-- demo_0_damages.csv
    |   |   |-- demo_0_header.txt
```

```
|   |   |-- demo_0_infernos.csv
|   |   |-- demo_0_kills.csv
|   |   |-- demo_0_smokes.csv
|   |   |-- demo_0_weapon_fires.csv
|   |   |-- demo_0.dem
|   |   `-- demo_0.zip
|   |-- ...
|   `-- log.txt
|-- ...
|-- partition_y/
|   |-- demo_x+n/
|   |   |-- demo_x+n_bomb.csv
|   |   |-- demo_x+n_damages.csv
|   |   |-- demo_x+n_header.txt
|   |   |-- demo_x+n_infernos.csv
|   |   |-- demo_x+n_kills.csv
|   |   |-- demo_x+n_smokes.csv
|   |   |-- demo_x+n_weapon_fires.csv
|   |   |-- demo_x+n.dem
|   |   `-- demo_x+n.zip
|   |-- ...
|   |-- demo_x+n+m/
|   |   |-- demo_x+n+m_bomb.csv
|   |   |-- demo_x+n+m_damages.csv
|   |   |-- demo_x+n+m_header.txt
|   |   |-- demo_x+n+m_infernos.csv
|   |   |-- demo_x+n+m_kills.csv
|   |   |-- demo_x+n+m_smokes.csv
|   |   |-- demo_x+n+m_weapon_fires.csv
|   |   |-- demo_x+n+m.dem
|   |   `-- demo_x+n+m.zip
|   `-- log.txt
`-- redundant/
    `-- .../
        `-- ...
```

Due to enginnering consideration, some redundant folders might be renamed with an addtional suffix `_0`, you can run the following scripts to rename them to the original form.

```
$ python redundant_folders_rename.py
> Enter the root directory path: <DEMO_ROOT_DIR>
> [printed info]
> Do you want to proceed with renaming? (y/n): y
> [printed info]
```

Next, label the cheaters within each match. The exceptions will be logged in `find_cheater_log.txt`.

```
$ python Find_Cheaters_in_Folder.py <DEMO_ROOT_DIR
    ↪ > <CHEATER_LIST_EXCEL_DIR>
```

Then your updated file structure should look like this. Each demo directory will now include a `cheaterlist.csv` file.

```
.
`-- demo_dir/
    |-- partition_0/
    |   |-- demo_0/
    |   |   |-- cheaterlist.csv
    |   |   |-- demo_0_bomb.csv
    |   |   |-- demo_0_damages.csv
    |   |   |-- demo_0_header.txt
    |   |   |-- demo_0_infernos.csv
    |   |   |-- demo_0_kills.csv
    |   |   |-- demo_0_smokes.csv
    |   |   |-- demo_0_weapon_fires.csv
    |   |   |-- demo_0.dem
    |   |   `-- demo_0.zip
    |   |-- ...
    |   `-- log.txt
    |-- ...
    |-- partition_y/
    |   |-- demo_x+n/
    |   |   |-- cheaterlist.csv
    |   |   |-- demo_x+n_bomb.csv
    |   |   |-- demo_x+n_damages.csv
    |   |   |-- demo_x+n_header.txt
    |   |   |-- demo_x+n_infernos.csv
    |   |   |-- demo_x+n_kills.csv
    |   |   |-- demo_x+n_smokes.csv
    |   |   |-- demo_x+n_weapon_fires.csv
    |   |   |-- demo_x+n.dem
    |   |   `-- demo_x+n.zip
    |   |-- ...
    |   |-- demo_x+n+m/
    |   |   |-- cheaterlist.csv
    |   |   |-- demo_x+n+m_bomb.csv
    |   |   |-- demo_x+n+m_damages.csv
    |   |   |-- demo_x+n+m_header.txt
    |   |   |-- demo_x+n+m_infernos.csv
    |   |   |-- demo_x+n+m_kills.csv
    |   |   |-- demo_x+n+m_smokes.csv
    |   |   |-- demo_x+n+m_weapon_fires.csv
    |   |   |-- demo_x+n+m.dem
    |   |   `-- demo_x+n+m.zip
    |   `-- log.txt
    `-- redundant/
        `-- .../
            `-- ...
```

[***Optional***] Some raw zip files might be corrupted during the download, you may use the following script to remove them.

```
$ python remove_corrupt_zips.py <DEMO_ROOT_DIR>
```

A.5.2 Trajectories Extraction

Run the following `shell` scripts to concurrently process the trajectory data. You should modify the directory and number of partitions based on your own data.

```
$ chmod +x run_3D_conversion_modified.sh
$ chmod +x run_2D_conversion_modified.sh
$ ./run_3D_conversion_modified.sh
$ ./run_2D_conversion_modified.sh
```

The logs (if any) for 2D/3D trajectory converters will be stored as `2D_trajectory_modified.txt` and `3D_trajectory_modified.txt`, respectively. The file structure should then look like this.

```
.
`-- demo_dir/
    |-- partition_0/
    |   |-- demo_0/
    |   |   |-- 2D_trajectory_modified.csv
    |   |   |-- 3D_trajectory_modified.csv
    |   |   |-- cheaterlist.csv
    |   |   |-- demo_0_bomb.csv
    |   |   |-- demo_0_damages.csv
    |   |   |-- demo_0_header.txt
    |   |   |-- demo_0_infernos.csv
    |   |   |-- demo_0_kills.csv
    |   |   |-- demo_0_smokes.csv
    |   |   |-- demo_0_weapon_fires.csv
```

```
|   |   |-- demo_0.dem
|   |   `-- demo_0.zip
|   |-- ...
|   `-- log.txt
|-- ...
|-- partition_y/
|   |-- demo_x+n/
|   |   |-- 2D_trajectory_modified.csv
|   |   |-- 3D_trajectory_modified.csv
|   |   |-- cheaterlist.csv
|   |   |-- demo_x+n_bomb.csv
|   |   |-- demo_x+n_damages.csv
|   |   |-- demo_x+n_header.txt
|   |   |-- demo_x+n_infernos.csv
|   |   |-- demo_x+n_kills.csv
|   |   |-- demo_x+n_smokes.csv
|   |   |-- demo_x+n_weapon_fires.csv
|   |   |-- demo_x+n.dem
|   |   `-- demo_x+n.zip
|   |-- ...
|   |-- demo_x+n+m/
|   |   |-- 2D_trajectory_modified.csv
|   |   |-- 3D_trajectory_modified.csv
|   |   |-- cheaterlist.csv
|   |   |-- demo_x+n+m_bomb.csv
|   |   |-- demo_x+n+m_damages.csv
|   |   |-- demo_x+n+m_header.txt
|   |   |-- demo_x+n+m_infernos.csv
|   |   |-- demo_x+n+m_kills.csv
|   |   |-- demo_x+n+m_smokes.csv
|   |   |-- demo_x+n+m_weapon_fires.csv
|   |   |-- demo_x+n+m.dem
|   |   `-- demo_x+n+m.zip
|   `-- log.txt
`-- redundant/
    `-- .../
        `-- ...
```

A.6 XGUARDIAN General Usage

This section describes the general usage of XGUARDIAN's core detection and explanation components.

A.6.1 INSPECTOR

Activate the conda environment `detector_clean`. Run `XGuardian-Inspector.py` with the following command, where `<train_path>`, `<val_path>`, `<test_path>` are the directories of the data splits. According to our paper findings, `<window_size>` should be `6`, `<stride>` should be `1`. `<drop_columns>` is the parameter to drop some redundant columns in the dataset. This should be set as `"isCheater,steamid,kill_num,match_id,in_crouch,is_walking,is_strafing,weapon,angle,x,y,distance"`. Note that at Line 82-83, the GPUs are designated; please adjust according to your device configurations.

```
$ python XGuardian-Inspector.py <train_path> <
    ↪ val_path> <test_path> <output_dir> <
    ↪ window_size> <stride> <drop_columns>
```

[*Directories of the datasets*] Except the detection performance (Appendix A.7.1) and elimination window size comparison (Appendix A.7.3), the `<train_path>`, `<val_path>`, `<test_path>` are as follows. The directories of the above two sections are explicitly stated in the relevant sections.

```
<train_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/Design/
    ↪ dataset/Train_cleaned.csv
<val_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/Design/
    ↪ dataset/Validation_cleaned.csv
<test_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/Design/
    ↪ dataset/Test_cleaned.csv
```

After execution, three CSVs will be generated named as `aggregated_preditions_x.csv`, where x is corresponding to training, validation or testing set, respectively. These CSVs should be first processed with the following commands for some of the following experiments. `<input_csv_path>` should be the directories of the `aggregated_preditions_x.csv`. `aggregated_preditions_x.csv` is the raw output for each elimination event, we use as an intermediate data it for processing the match predictions

```
$ python XGuardian-Inspector-Match-Preprocessing.
    ↪ py <input_csv_path> <output_csv_path>
```

Then run `XGuardian-Inspector-Match.py` for the final detection result. In which, `<feature_selection>` can be set as `'o'` (original, please use this one normally), `'b'` (binary), and `'a'` (all, original+binary). For details, please refer to Section 5.4.5. Note that at Line 46-47, the GPUs are designated; please adjust according to your device configurations.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path <training_path> --val_path <
    ↪ validation_path> --test_path <test_path> --
    ↪ feature_set <feature_selection> --
    ↪ output_dir <output_directory>
```

In the output directory, `round_integ_x_predictions_n.csv` will be generated, where x is corresponding to training, validation or testing set; n is corresponding to original, binary and all (feature selection). The above CSV is the final detection results of XGUARDIAN, you may use `round_eval_metrics_stats.py` to evaluate the metrics. `round_integ_x_predictions_n.csv` is the final prediction results of XGUARDIAN for each set, `original` means we are applying `'o'` settings for predictions stage embedding style comparison. **[*Important note*]** You should always manually change the directory of `round_integ_x_predictions_n.csv` in `round_eval_metrics_stats.py` at Line 5 before using it.

```
$ python round_eval_metrics_stats.py
```

A.6.2 EXPLAINER

Activate the conda environment `explainer_clean`. Run `XGuardian-Explainer-Elimination.py` to generate raw

SHAP values per elimination. This script's usage is very similar to XGuardian-Inspector.py. <window_size> should be 6, <stride> should be 1. <drop_columns> is the parameter to drop some redundant columns in the dataset. This should be set as "isCheater,steamid,kill_num, match_id,in_crouch,is_walking,is_strafing, weapon,angle,x,y,distance". This script will not train the detection model again, but will use the trained models (check Line 270), so please make sure the model directories are correct.

```
$ python XGuardian-Explainer.py <train_path> <
    ↪ val_path> <test_path> <output_dir> <
    ↪ window_size> <stride> <drop_columns>
```

The above execution will generate shap_values _with_ids.npz. Then run XGuardian-Explainer-Elimination-xguardian_value_avg.py to generate the XGUARDIAN values per elimination. You should manually change the input directories: csv_file_path (Line 5, your test set directory, i.e., Test_cleaned.csv) and shap_values (Line 10, your shap_values_with_ids.npz directory), and output_csv_file_path (Line 81).

```
$ python XGuardian-Explainer-Elimination-
    ↪ xguardian_value_avg.py
```

For elimination visualization, please check /home2 /data1/J.Zhang/XGuardian/XGuardian/Design/src /Explainer_vis folder. There are some preprocessing needs to be done. Run split_shap_csv_to_subCSVs.py. At Line 38-39, you should change the input (the XGuardian-Explainer-Elimination-xguardian_value_avg.py generated CSV) and output directories.

```
$ python split_shap_csv_to_subCSVs.py
```

Then it will generate a subfolder test_set_data. Run 2D _vis.sh to concurrently visualize the trajectory. In which, at Line 4-5, change the input and output directories. Also, at Line 22, make sure the Python script directory is correct (using XGuardian/XGuardian/Design/src/Explainer_vis /2D_vis_batch_concurrent.py). At Line 30-31, you should adjust the total number of the cheaters and normal players (check the numbering in test_set_data, no need to replace if using our dataset). Due to a different coordination system, the final visualizations (PNGs and MP4s) need to be center-rotated 180 degrees to match the user's aiming trajectories.

```
$ ./2D_vis.sh
```

For match EXPLAINER visualization, run XGuardian-Explainer-Match.py. <model_path> is the directory of logistic_regression_model_original.joblib (this is the model of Match-Inspector, just ignore the naming, we are actually using random forest).

```
$ python XGuardian-Explainer-Match.py --
    ↪ dataset_path <test_set_directory> --
    ↪ train_path <training_set_directory> --
    ↪ model_path <model_path> --feature_set <
    ↪ feature_selection> --output_dir <
    ↪ output_directory>
```

A.7 Experiments

This subsection contains the instructions to run and check the raw results of each experiment in our paper. There are some redundant files for debugging and testing, which can be ignored for reusing/reproducing XGUARDIAN. Please follow the instructions below to reuse/reproduce XGUARDIAN. After v2, each experiment folder contains its own round_eval _metrics_stats.py for evaluation, please ignore all firectory modification instructions in the following section.

A.7.1 Detection Performance

[*To check the raw results*] Check each round_integ _test_predictions_original.csv under each subfolder of /home2/data1/J.Zhang/XGuardian/XGuardian /Experiments/5.2_Detection_Performace. Use round _eval_metrics_stats.py to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[*To reproduce*] Run the following command by replacing the order of the train, validation, and test sets in the dataset thoroughly with the conda environment detector_clean activated.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector.py <
    ↪ train_path> <val_path> <test_path> <
    ↪ output_dir> 6 1 "isCheater,steamid,kill_num
    ↪ ,match_id,in_crouch,is_walking,is_strafing,
    ↪ weapon,angle,x,y,distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_train.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_val.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_test.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

[***Directories***] In this experiments, the directories of would be depending on which set combination is applied to verify the results. For example if now testing the combination of `TestTrainVal`, the directories of `<train_path>`, `<val_path>`, `<test_path>` should be:

```
<train_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/Design/
    ↪ dataset/Test_cleaned.csv
<val_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/Design/
    ↪ dataset/Train_cleaned.csv
<test_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/Design/
    ↪ dataset/Validation_cleaned.csv
```

All of the results can be found at `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.2_Detection_Performace/overall_performance/xxx/round_integ_test_predictions_original.csv`, where `xxx` are `TrainValTest`, `TrainTestVal`, `ValTrainTest`, `ValTestTrain`, `TestTrainVal`, `TestValTrain`. Please then use `/home2/data1/J.Zhang/XGuardian/XGuardian/Design/src/round_eval_metrics_stats.py` to evaluate each set combination results (remember to change different directory at Line 5 in `round_eval_metrics_stats.py`). The returned results from `round_eval_metrics_stats.py` would be the results of Section 5.2.

A.7.2 Feature Ablation Study

[***To check the raw results***] Check each `round_integ_test_predictions_original.csv` under each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.3_Feature_Ablation_Study/feature_selection`. Use `round_eval_metrics_stats.py` to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[***To reproduce***] Run the following command by modifying the removed columns parameter from the dataset with the conda environment `detector_clean` activated. As default, `<drop_columns>` would be `"isCheater,steamid,kill_num,match_id,in_crouch,is_walking,is_strafing,weapon,angle,x,y,distance"`. This is the case `all`, where all of the features are included. When doing the ablation study, one might just add the feature that would like to be ablated in `<drop_columns>`. For example, when removing angular change, add `angle_change` to the default parameter. Other valid options to be added are `velocity_x,velocity_y,acceleration_x,acceleration_y`.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector.py XGuardian
    ↪ /XGuardian/Design/dataset/Train_cleaned.csv
    ↪  XGuardian/XGuardian/Design/dataset/
    ↪ Validation_cleaned.csv XGuardian/XGuardian/
    ↪ Design/dataset/Test_cleaned.csv <output_dir
    ↪ > 6 1 <drop_columns>
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_train.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_val.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_test.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian-Inspector-
    ↪ Match.py --train_path <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

A.7.3 Elimination Window Comparison

[***To check the raw results***] Check each `round_integ_test_predictions_original.csv` under each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.4.1_Different_Elimination_Window_Sizes/diff_window_comp/results`. Use `round_eval_metrics_stats.py` to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[***To reproduce***] Run the following commands with the conda environment `detector_clean` activated.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/src/XGuardian_128.py
    ↪ XGuardian/XGuardian/Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/ready_for_eval_data/128
    ↪ _train.csv XGuardian/XGuardian/Experiments/
    ↪ 5.4.1_Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/ready_for_eval_data/128
    ↪ _val.csv XGuardian/XGuardian/Experiments/
    ↪ 5.4.1_Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/ready_for_eval_data/128
    ↪ _test.csv <output_dir> 6 1 "isCheater,
    ↪ steamid,kill_num,match_id,in_crouch,
    ↪ is_walking,is_strafing,weapon,angle,x,y,
    ↪ distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/src/XGuardian_256.py
    ↪ XGuardian/XGuardian/Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
```

```
    ↪ diff_window_comp/ready_for_eval_data/256
    ↪ _train.csv XGuardian/XGuardian/Experiments/
    ↪ 5.4.1_Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/ready_for_eval_data/256
    ↪ _val.csv XGuardian/XGuardian/Experiments/
    ↪ 5.4.1_Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/ready_for_eval_data/256
    ↪ _test.csv <output_dir> 6 1 "isCheater,
    ↪ steamid,kill_num,match_id,in_crouch,
    ↪ is_walking,is_strafing,weapon,angle,x,y,
    ↪ distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_train.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_val.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_test.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

[*Directories*] The directories of <train_path>, <val_path>, <test_path> for elimination window size = 384(-256/+128) should be:

```
<train_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/original_data/2
    ↪ D_tra_vel_acc_train.csv
<val_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/original_data/2
    ↪ D_tra_vel_acc_val.csv
<test_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/original_data/2
    ↪ D_tra_vel_acc_test.csv
```

The directories of <train_path>, <val_path>, <test_path> for elimination window size = 192(-128/+64) should be:

```
<train_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/original_data/128_train.
    ↪ csv
<val_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/original_data/128_val.csv,
<test_path>:
/home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.1
    ↪ _Different_Elimination_Window_Sizes/
    ↪ diff_window_comp/original_data/128_test.csv
    ↪ ,
```

[*Naming*] You may have noticed that the naming for the above datasets are different from the main datasets. The file with _cleaned suffix means it was generated by the INSPECTOR. It filters out some edge cases mentioned in Section 5.1.1 - Data Cleansing of our paper and also removes some redundant columns.

A.7.4 Sliding Window Comparison

[*To check the raw results*] Check each round_integ_test_predictions_original.csv under each subfolder of /home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.4.2_Different_Sliding_Window_Sizes/diff_sliding_window_comp. Use round_eval_metrics_stats.py to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[*To reproduce*] Run the following commands with the conda environment detector_clean activated. Modify sliding window parameter <window_size> accordingly.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector.py XGuardian
    ↪ /XGuardian/Design/dataset/Train_cleaned.csv
    ↪  XGuardian/XGuardian/Design/dataset/
    ↪ Validation_cleaned.csv XGuardian/XGuardian/
    ↪ Design/dataset/Test_cleaned.csv <output_dir
    ↪ > <window_size> 1 "isCheater,steamid,
    ↪ kill_num,match_id,in_crouch,is_walking,
    ↪ is_strafing,weapon,angle,x,y,distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_train.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_val.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_test.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv
```

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

A.7.5 Different Model Comparison in Embedding Generation

[*To check the raw results*] Check each `round_integ_test_predictions_original.csv` under each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.4.3_Different_Models_in_Embedding_Generation/embedding`. Use `round_eval_metrics_stats.py` to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[*To reproduce*] BiLSTM and Transformer can be directly executed under `detector_clean` environment with the following commands, respectively.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.3
    ↪ _Different_Models_in_Embedding_Generation/
    ↪ embedding/BiLSTM.py XGuardian/XGuardian/
    ↪ Design/dataset/Train_cleaned.csv XGuardian/
    ↪ XGuardian/Design/dataset/Validation_cleaned
    ↪ .csv XGuardian/XGuardian/Design/dataset/
    ↪ Test_cleaned.csv <output_dir> 6 1 "
    ↪ isCheater,steamid,kill_num,match_id,
    ↪ in_crouch,is_walking,is_strafing,weapon,
    ↪ angle,x,y,distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.3
    ↪ _Different_Models_in_Embedding_Generation/
    ↪ embedding/Transformer.py XGuardian/
    ↪ XGuardian/Design/dataset/Train_cleaned.csv
    ↪ XGuardian/XGuardian/Design/dataset/
    ↪ Validation_cleaned.csv XGuardian/XGuardian/
    ↪ Design/dataset/Test_cleaned.csv <output_dir
    ↪ > 6 1 "isCheater,steamid,kill_num,match_id,
    ↪ in_crouch,is_walking,is_strafing,weapon,
    ↪ angle,x,y,distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_train.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_val.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_test.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

Clone the environment `detector_clean` and install `keras-tcn` (https://github.com/philipperemy/keras-tcn). A segement of the code (from Line 183 to Line 221 was made as comments during the debugging and testing phases, please cancel the comment hashtags while training).

```
$ conda create --name tcn --clone detector_clean
$ conda activate tcn
$ pip install keras-tcn
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.3
    ↪ _Different_Models_in_Embedding_Generation/
    ↪ embedding/TCN.py XGuardian/XGuardian/Design
    ↪ /dataset/Train_cleaned.csv XGuardian/
    ↪ XGuardian/Design/dataset/Validation_cleaned
    ↪ .csv XGuardian/XGuardian/Design/dataset/
    ↪ Test_cleaned.csv <output_dir> 6 1 "
    ↪ isCheater,steamid,kill_num,match_id,
    ↪ in_crouch,is_walking,is_strafing,weapon,
    ↪ angle,x,y,distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_train.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_val.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_test.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

A.7.6 Different Model Comparison in Trajectory Classification

[*To check the raw results*] Check each `round_integ_test_predictions_original.csv` under each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.4.4_Different_Models_in_Trajectory_Classification/integration`. Use `round_eval_metrics_stats.py` to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[*To reproduce*] Run the following command with the conda environment `detector_clean` activated.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.4.4
    ↪ _Different_Models_in_Trajectory_Classification
    ↪ /integration/GRU-fc.py XGuardian/XGuardian/
    ↪ Design/dataset/Train_cleaned.csv XGuardian/
    ↪ XGuardian/Design/dataset/Validation_cleaned
    ↪ .csv XGuardian/XGuardian/Design/dataset/
    ↪ Test_cleaned.csv <output_dir> 6 1 "
    ↪ isCheater,steamid,kill_num,match_id,
    ↪ in_crouch,is_walking,is_strafing,weapon,
    ↪ angle,x,y,distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_train.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_val.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py <path_of_each_subfolder>/
    ↪ aggregated_predictions_test.csv <
    ↪ path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path <path_of_each_subfolder>/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

A.7.7 Different Input Style Comparison in Elimination Prediction

[*To check the raw results*] Check each `round_integ_test_predictions_original.csv` under each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.4.5_Input_Styles_in_Elimination_Predictions`. Use `round_eval_metrics_stats.py` to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[*To reproduce*] Run `XGuardian-Inspector-Match.py` for the final detection result. In which, `<feature_selection>` can be set as `'o'` (original, please use this one normally), `'b'` (binary), and `'a'` (all, original+binary).

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path XGuardian/XGuardian/Experiments/
    ↪ 5.2_Detection_Performace/
    ↪ overall_performance/TrainValTest/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path XGuardian/XGuardian/Experiments/
    ↪ 5.2_Detection_Performace/
    ↪ overall_performance/TrainValTest/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path XGuardian/XGuardian/Experiments/
    ↪ 5.2_Detection_Performace/
    ↪ overall_performance/TrainValTest/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set <feature_selection> --
    ↪ output_dir <output_directory>
```

A.7.8 Comparative Study

[*To check the raw results*] In terms of HAWK. Check each `round_integ_test_predictions_original.csv` under each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.5_Comparative_Study`. Use `round_eval_metrics_stats.py` to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

In terms of statistical methods. Run the following commands.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/
    ↪ stats_method/AccA_eval.py
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/
    ↪ stats_method/HitAcc_eval.py
```

[*To reproduce*] In terms of HAWK. Run the following commands. Due to minimizing the size of our artifact, we deleted some intermediate files (e.g., `aggregated_prediction_x.csv`, etc.), you may reproduce it by using `XGuardian-Inspector.py`.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/hawk/
    ↪ hawk_xguardian.py XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/hawk/
    ↪ HAWK_train.csv XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/hawk/
    ↪ HAWK_validation.csv XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/hawk/
    ↪ HAWK_test.csv <output_dir> 6 1 "isCheater,
    ↪ steamid,kill_num,match_id,in_crouch,
    ↪ is_walking,is_strafing,weapon,angle,x,y,
    ↪ distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/hawk/
```

```
    ↪ aggregated_predictions_train.csv XGuardian/
    ↪ XGuardian/Experiments/5.5_Comparative_Study
    ↪ /hawk/aggregated_predictions_train_cleaned.
    ↪ csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/hawk/
    ↪ aggregated_predictions_val.csv XGuardian/
    ↪ XGuardian/Experiments/5.5_Comparative_Study
    ↪ /hawk/aggregated_predictions_val_cleaned.
    ↪ csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/hawk/
    ↪ aggregated_predictions_test.csv XGuardian/
    ↪ XGuardian/Experiments/5.5_Comparative_Study
    ↪ /hawk/aggregated_predictions_test_cleaned.
    ↪ csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path XGuardian/XGuardian/Experiments/
    ↪ 5.5_Comparative_Study/hawk/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ val_path XGuardian/XGuardian/Experiments/
    ↪ 5.5_Comparative_Study/hawk/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path XGuardian/XGuardian/Experiments/
    ↪ 5.5_Comparative_Study/hawk/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --output_dir <
    ↪ output_directory>
```

In terms of statistical methods. Run the following commands.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/
    ↪ stats_method/AccA_eval.py
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.5_Comparative_Study/
    ↪ stats_method/HitAcc_eval.py
```

A.7.9 Generalizability

[*To check the raw results*] Check each `round_integ_test_predictions_original.csv` under each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.6_Generalizability/farlight84/result/mobile`. Use `round_eval_metrics_stats.py` to calculate the metrics.

```
$ python round_eval_metrics_stats.py
```

[*To reproduce*] Run the following commands. Due to minimizing the size of our artifact, we deleted some intermediate files (e.g., `aggregated_prediction_x.csv`, etc.), you may reproduce it by using `XGuardian-Inspector.py`. For HAWK's results and reproducible procedures, please check the previous subsubsection.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.6_Generalizability/farlight84
    ↪ /src/XGuardian/XGuardian-Inspector.py
    ↪ XGuardian/XGuardian/Experiments/5.6
    ↪ _Generalizability/farlight84/dataset/mobile
    ↪ /Farlight84_Mobile_train.csv XGuardian/
    ↪ XGuardian/Experiments/5.6_Generalizability/
    ↪ farlight84/dataset/mobile/
    ↪ Farlight84_Mobile_val.csv XGuardian/
    ↪ XGuardian/Experiments/5.6_Generalizability/
    ↪ farlight84/dataset/mobile/
    ↪ Farlight84_Mobile_test.csv <output_dir> 6 1
    ↪  "isCheater,steamid,kill_num,match_id,angle
    ↪ ,x,y,platform"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py XGuardian/XGuardian/
    ↪ Experiments/5.6_Generalizability/farlight84
    ↪ /result/mobile/aggregated_predictions_train
    ↪ .csv XGuardian/XGuardian/Experiments/5.6
    ↪ _Generalizability/farlight84/result/mobile/
    ↪ aggregated_predictions_train_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py XGuardian/XGuardian/
    ↪ Experiments/5.6_Generalizability/farlight84
    ↪ /result/mobile/aggregated_predictions_val.
    ↪ csv XGuardian/XGuardian/Experiments/5.6
    ↪ _Generalizability/farlight84/result/mobile/
    ↪ aggregated_predictions_val_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match-
    ↪ Preprocessing.py XGuardian/XGuardian/
    ↪ Experiments/5.6_Generalizability/farlight84
    ↪ /result/mobile/aggregated_predictions_test.
    ↪ csv XGuardian/XGuardian/Experiments/5.6
    ↪ _Generalizability/farlight84/result/mobile/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Design/src/XGuardian-Inspector-Match.py --
    ↪ train_path XGuardian/XGuardian/Experiments/
    ↪ 5.6_Generalizability/farlight84/result/
    ↪ mobile/aggregated_predictions_train_cleaned
    ↪ .csv --val_path XGuardian/XGuardian/
    ↪ Experiments/5.6_Generalizability/farlight84
    ↪ /result/mobile/
    ↪ aggregated_predictions_val_cleaned.csv --
    ↪ test_path XGuardian/XGuardian/Experiments/
    ↪ 5.6_Generalizability/farlight84/result/
    ↪ mobile/aggregated_predictions_test_cleaned.
    ↪ csv --feature_set 'o' --output_dir <
    ↪ output_directory>
```

A.7.10 Case Study

[*To check the raw results*] Each subfolder of `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.7_Case_Study` contains the corresponding results.
[*To reproduce*] For elimination scope, please follow the procedures in Appendix A.6.2. For match scope, run the following command.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.7_Case_Study/match_scope/
    ↪ shap_case_Study.py --train_path XGuardian/
    ↪ XGuardian/Experiments/5.2
    ↪ _Detection_Performace/overall_performance/
    ↪ TrainValTest/
    ↪ aggregated_predictions_train_cleaned.csv --
    ↪ dataset_path XGuardian/XGuardian/
    ↪ Experiments/5.2_Detection_Performace/
```

```
    ↪ overall_performance/TrainValTest/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ model_path XGuardian/XGuardian/Experiments/
    ↪ 5.2_Detection_Performace/
    ↪ overall_performance/TrainValTest/
    ↪ logistic_regression_model_original.joblib
    ↪ --feature_set 'o' --output_dir <output_dir>
```

A.7.11 Robustness to Adversarial Attack

[*To check the raw results*] Use `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.8_Adversarial_Attack/src/round_eval_metrics_stats.py` to calculate the metrics. The directory of the file `round_integ_test_predictions_original.csv` must be modified before the execution at Line 5.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.8_Adversarial_Attack/src/
    ↪ round_eval_metrics_stats.py
```

[*To reproduce*] Run the following commands. Due to minimizing the size of our artifact, we deleted some intermediate files (e.g., `aggregated_prediction_x.csv`, etc.), you may reproduce it by using `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.8_Adversarial_Attack/src/XGuardian-Inspector-Prediction-only.py`. Note that the directories of the models `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.2_Detection_Performace/overall_performance/TrainValTest/gru_cnn_best_model.keras`, `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.2_Detection_Performace/overall_performance/TrainValTest/aggregation_best_model.keras` and `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.2_Detection_Performace/overall_performance/TrainValTest/best_aggregation_threshold_with_attention.txt` should be properly modified at Line 187-190, Line 256-260 and Line 263-268, respectively. Simply, you may instead copy these models to the `<output_dir>` directly.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.8_Adversarial_Attack/src/
    ↪ XGuardian-Inspector-Prediction-only.py
    ↪ XGuardian/XGuardian/Experiments/5.8
    ↪ _Adversarial_Attack/data/
    ↪ adversarial_dataset.csv XGuardian/XGuardian
    ↪ /Experiments/5.8_Adversarial_Attack/result
    ↪ 6 1 "isCheater,steamid,kill_num,match_id,
    ↪ in_crouch,is_walking,is_strafing,weapon,
    ↪ angle,x,y,distance"
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.8_Adversarial_Attack/src/
    ↪ XGuardian-Inspector-Match-Preprocessing.py
    ↪ XGuardian/XGuardian/Experiments/5.8
    ↪ _Adversarial_Attack/result/
    ↪ aggregated_predictions_test.csv XGuardian/
    ↪ XGuardian/Experiments/5.8
    ↪ _Adversarial_Attack/result/
    ↪ aggregated_predictions_test_cleaned.csv
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.8_Adversarial_Attack/src/
    ↪ XGuardian-Inspector-Match-Prediction-only.
    ↪ py --test_path XGuardian/XGuardian/
    ↪ Experiments/5.8_Adversarial_Attack/result/
    ↪ aggregated_predictions_test_cleaned.csv --
    ↪ feature_set 'o' --model_path XGuardian/
    ↪ XGuardian/Experiments/5.8
    ↪ _Adversarial_Attack/result/
    ↪ logistic_regression_model_original.joblib
    ↪ --output_dir XGuardian/XGuardian/
    ↪ Experiments/5.8_Adversarial_Attack/result
```

[*Reuse of* INSPECTOR *without training*] Please note that the above three scripts can be used to reuse the pre-trained model from XGUARDIAN, solely for performing prediction functions.

A.7.12 Overhead

[*To check the raw results*] Please check `data preprocessing_overhead.log`, `initialmodeltraining_overhead.log` and `testing_overhead.log` at `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.9_Overhead/overhead`.
[*To reproduce*] Run the following command with `detector_cleaned` environment.

```
$ python /home2/data1/J.Zhang/XGuardian/XGuardian/
    ↪ Experiments/5.9_Overhead/overhead/
    ↪ GRU_aggregated_2D_feature_selection.py
    ↪ XGuardian/XGuardian/Experiments/5.9
    ↪ _Overhead/overhead/Train_cleaned.csv
    ↪ XGuardian/XGuardian/Experiments/5.9
    ↪ _Overhead/overhead/Validation_cleaned.csv
    ↪ XGuardian/XGuardian/Experiments/5.9
    ↪ _Overhead/overhead/Test_cleaned.csv <
    ↪ output_dir> 6 1 "isCheater,steamid,kill_num
    ↪ ,match_id,in_crouch,is_walking,is_strafing,
    ↪ weapon,angle,x,y,distance"
```

A.7.13 User Study

The original demos are stored at `/home2/data1/J.Zhang/XGuardian/XGuardian/Experiments/5.10_User_Study_(addtional_evaluation)`. Please check our paper for the answer of the questionnaires and the operation procedures.

A.8 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/usenixsec2026/.





USENIX Security '26 Artifact Appendix: Send and Pretend: Exploiting Transcript Consistency Issues in End-to-End Encrypted Group Chats

Gabriel K. Gegenhuber[1], Moritz Grefner[2], Maximilian Günther[3], Matthäus Wininger[2], David Schmidt[2,4,5], and Aljosha Judmayer[2]

[1]Interdisciplinary Transformation University (IT:U), [2]University of Vienna, Faculty of Computer Science, [3]SBA Research, [4]UniVie Doctoral School Computer Science, [5]CDL AsTra

A Artifact Appendix

A.1 Abstract

This artifact contains the proof-of-concept code used to demonstrate transcript consistency issues in group chats of WhatsApp, Signal, iMessage, and Threema. More specifically, current end-to-end-encryption designs in group chats allow a malicious group participant to send different messages to different group members. The codebase includes modified clients that send inconsistent group messages, and manipulate poll creation, voting, and closing messages. Further, it contains code to demonstrate implementation-specific issues such as fabricated quoted messages and inconsistent duplicate handling.

A.2 Description & Requirements

Since our PoCs have to interact with four different messaging services, we have to use four different client approaches and hence four different software stacks. The most streamlined and easiest to reproduce is probably the Signal-related workflow, as it is completely dockerized. Therefore, we would recommend the evaluator to start there.

A.2.1 Security, Privacy, and Ethical Concerns

Unfortunately we cannot provide test accounts for all respective instant messaging services. We encourage the evaluators to use dedicated test accounts if available instead of their own private accounts. This should prevent accidental exposure of private (and potentially sensitive) inbound instant messages to less secure environments such as the evaluators' test setups configured as linked devices. During testing, we never experienced any countermeasures by the vendors that would result in a block or ban of the involved accounts. This might be due to the fact that (on the server side) most of the TC attacks are indistinguishable from normal messaging behavior.

A.2.2 How to Access

We provide our code on GitHub at `https://github.com/sbaresearch/transcript-consistency` and on Zenodo for long-term archiving at `https://doi.org/10.5281/zenodo.20323807`.

A.2.3 Hardware Dependencies

A working Android and/or iOS device running the according messaging app (primary device) is required to test the described transcript consistency and implementation issues for the respective messaging service.

A.2.4 Software Dependencies

The exact software dependencies are listed in the dependency files in the corresponding artifact subdirectories. To provide a high-level overview, the artifacts require:

- Git and standard build tools for the host operating system.
- Go tooling for the WhatsApp clients based on `whatsmeow`.
- Rust tooling for the iMessage client based on `rustpush`.
- Python and Docker for the Signal client containers based on `Signal-Desktop`.
- Android Studio, the Android SDK, the Android NDK, `bash`, the Protobuf compiler, the Rust toolchain, and a Java/Kotlin build environment for the modified Threema Android debug client, or only Docker for building a release APK using the provided build script.
- Official messenger apps for the receiver devices used in the experiments.

The artifacts do not include credentials, phone numbers, Apple IDs, or Threema licenses. Therefore, reproducing the results requires creating the respective accounts.

A.2.5 Benchmarks

None

A.3 Setup

A.3.1 Installation

Download the archive using the provided link and extract the content to disk. The extracted content should look like this:

- `videos/`: This directory contains PoC videos showing the behavior of official clients in a group with a malicious client.
- `code/`: This directory contains the code of our custom clients for all targeted messengers.
- `code/signal/poc-skdm/`: This directory contains a modified Signal Desktop [2] client demonstrating the Sender Key with E2EE fallback case for a specific recipient selected in the GUI.
- `code/signal/poc-groups/`: This directory contains a modified Signal Desktop [2] client supporting server- and client-fan-out-only modes and providing a more advanced GUI for exclusions and overrides for all considered message types (e.g., polls, regular messages).
- `code/imessage/`: This directory contains a modified version of rustpush [1] that includes a command-line client to send inconsistent messages into group chats and to manipulate polls and votes.
- `code/threema/threema-android/`: This directory contains a modified Threema Android [3] client with support for commands in relevant input fields (for regular messages and polls) to control excluded recipients and overrides, as well as poll voting and closing.
- `code/whatsapp/poc-client/`: This directory contains our modified WhatsApp client that allows sending inconsistent messages while supporting server- and client-fan-out-only modes, as well as Sender Key with E2EE fallback, which builds on whatsmeow [4].
- `code/whatsapp/quote-client/`: This directory contains our modified WhatsApp client to manipulate quotes, building on whatsmeow [4].

Throughout the repository, `README.md` files provide the information required to reproduce the results.

A.3.2 Basic Test

For the Signal Desktop client, in particular, we provide containers for improved reproducibility since there are many dependencies and the build steps would otherwise be prone to failure. The entire GUI can be run inside a container.

Therefore, switch into either of the `code/signal/{poc-skdm,poc-groups}` directories and execute `./build-container.py` followed by `./start-container.py`. This fire up a container with VNC installed together with a browser from which it can be accessed via `http://localhost:6080/`. In this container the modified Signal desktop can be built and executed with `just run`. Do note that Signal user data will be stored in a Docker volume that has to be manually removed afterwards.

A.4 Evaluation Workflow

A.4.1 Major Claims

(C1): WhatsApp, Signal, iMessage, and Threema do not have transcript consistency in group chats. A malicious group participant can send malicious messages to selected participants without triggering server-side detection or clear recipient-facing UI warnings. This result corresponds to Sections 4 and 5.1, Table 2, and Experiments E1–E3.

(C2): Higher-level interactive group features, in particular polls, inherit and amplify the transcript consistency problem. A malicious poll creator or voter can make honest participants observe divergent poll questions, options, intermediate results, or final results. This corresponds to Section 5.2.3 and Figures 9 and 10, and is evaluated by Experiment E4.

(C3): Implementation-specific behavior further facilitates practical exploitation. This includes fabricated quoted messages in WhatsApp and Signal, spoofed or misleading vote attribution in iMessage, platform-specific WhatsApp poll parsing differences, Signal duplicate handling differences, and WhatsApp OS/device fingerprinting through message IDs. This corresponds to Section 5.3, Table 3, and Appendix A.2 of the paper, and is evaluated by Experiment E5.

A.4.2 Experiments

(E1): Pairwise group message equivocation [45 human-minutes + 20 compute-minutes per messenger]: This experiment reproduces the group transcript inconsistency vector for pairwise E2EE delivery. It applies directly to iMessage and Threema.
Preparation: Create a test group with one attacker account and at least two honest recipients. Build and authenticate the modified client for the selected messenger.
Execution: Use the modified client to send message `M1` to one recipient and message `M2` to another recipient under the same group conversation context. For omission tests, exclude one recipient entirely.
Results: The receiving official clients should show different local transcripts: one recipient sees `M1`, another

sees `M2`, and excluded recipients do not see the message. The clients should not display a clear warning that the group transcript diverged.

(E2): Sender Key fallback and retransmission [45 human-minutes + 20 compute-minutes per messenger]: This experiment reproduces the WhatsApp and Signal vector in which Sender Key delivery failure or retry behavior leads to pairwise retransmission and enables recipient-specific content. For Signal, this experiment requires using the `poc-skdm` client in contrast to all other experiments, which can be built and started in the same way as the `poc-groups` version.

Preparation: Create WhatsApp or Signal test groups and build the corresponding modified client. Ensure that at least two receiving devices are online.

Execution: Use the artifact's Sender Key distribution or fallback controls to cause one recipient to require a retry or retransmission. Respond with a recipient-specific replacement message. For WhatsApp, also evaluate the configuration that hides the transient decrypt-fail UI. For Signal, use the provided recipient selection UI to determine the recipient which will receive the modified message. The client will append hardcoded text.

Results: The targeted recipient should receive the replacement message while other participants retain the original message or no corresponding message. Signal should fail silently; WhatsApp's transient decrypt-fail indicator should disappear after retransmission or be suppressed where the artifact exercises the corresponding attribute.

(E3): Sender Key-only equivocation [45 human-minutes + 20 compute-minutes per messenger]: This experiment reproduces equivocation using Sender Key group messages without delivering the final equivocated content through pairwise sessions.

Preparation: Use the WhatsApp or Signal modified client in server fan-out-only mode and create a test group with at least two recipients.

Execution: Broadcast different versions of a group message so that clients that already accepted one message version ignore later transmissions with the same message identity, while selected recipients receive a different version.

Results: Honest participants should end with divergent transcripts even though the attack uses Sender Key group-message delivery. No participant should receive a clear warning that a contradictory message version was sent to the group.

(E4): Poll manipulation [45 human-minutes + 20 compute-minutes per messenger]: This experiment reproduces the poll attacks from Section 5.2.3.

Preparation: Create a group whose clients support polls. Build the corresponding modified client and create a benign poll to verify normal poll rendering.

Execution: As a malicious poll creator, send different poll questions, option orders, option identifiers, or option text variants to different recipients. As a malicious voter, send different vote messages to different recipients or, for Threema and iMessage, use the service-specific vote manipulation supported by the artifact.

Results: Recipients should observe different poll questions, options, intermediate results, or final results. For index- or identifier-based schemes, option reordering should make a victim's intended vote count for a different option elsewhere. For WhatsApp, insignificant option changes or inverted questions should produce divergent interpretations. For Threema, manipulated close messages can overwrite final results.

(E5): Implementation-specific issues [45 human-minutes + 20 compute-minutes]: This experiment reproduces the additional implementation-specific findings.

Preparation: Select the relevant sub-artifact from the WhatsApp quote client, the Signal modified client, or the iMessage client. Prepare the official receiving clients.

Execution: For quoted-message tests, send a reply that references a missing or future message and supplies attacker-controlled fallback text. For iMessage, send a poll vote JSON with manipulated `participantHandle` values. For Signal, run the duplicate-handling examples and compare them with the supplied videos. If possible, additionally compare WhatsApp poll parsing and message IDs across Android, iOS, Web, Windows, and macOS clients.

Results: WhatsApp and Signal should display attacker-controlled quote fallback text for missing references, with WhatsApp providing no clear warning. iMessage should show misleading vote attribution in the poll preview while the detailed view remains more restrictive. WhatsApp clients should differ in poll parsing behavior across platforms, and WhatsApp message IDs should reveal OS/device-specific prefixes or lengths as described in Table 3.

A.5 Notes on Reusability

Reproducing the exploits in and end-to-end manner requires both multiple accounts (i.e., phone numbers or licenses) on different messaging platforms and different smartphones (i.e., Android, iOS). Unfortunately, we cannot provide this in a scalable way (i.e., for multiple reviewers). As a replacement, we provide pre-recorded videos of the exploits in practice in addition to the PoC source code.

Disclaimer: We are not aware of platform operators detecting and taking action against the provided exploits. However, we strongly recommend only executing the PoC with specific test accounts.

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/usenixsec2026/.

References

[1] GitHub – Rustpush. Accessed: 2026-01-27. URL: https://github.com/OpenBubbles/rustpush.

[2] GitHub – Signal Desktop. Accessed: 2026-01-27. URL: https://github.com/signalapp/Signal-Desktop.

[3] GitHub – Threema for Android. Accessed: 2026-01-27. URL: https://github.com/threema-ch/threema-android.

[4] GitHub – whatsmeow. Accessed: 2026-01-27. URL: https://github.com/tulir/whatsmeow.



USENIX Security '26 Artifact Appendix: Relay and Betray: Exploiting Client-Side Authority in Multi-User Mixed Reality

Mutahar Ali
University of California, Irvine

Habiba Farrukh
University of California, Irvine

A Artifact Appendix

A.1 Abstract

Our artifact supports the paper's security analysis of client-authoritative multi-user mixed reality (MR) apps. It is divided into two components: a public component and a private component.

A.1.1 Public Component

Publicly archived on Zenodo as two files: *relay-and-betray.zip*, the complete toolchain, and *videos.zip*, the recorded video demonstrations of the attacks. The toolchain has five parts:

1. **Static analysis.** A patched Il2CppDumper plus headless Ghidra scripts that recover class hierarchies, method signatures, and field names from stripped Unity IL2CPP binaries.
2. **Gadget injection.** A Frida gadget-injection toolchain for non-rooted Android-based devices, handling split APK sets, Google PairIP license checks, and `TracerPid` anti-debug checks, plus the device scripts that pull, patch, reinstall, and attach to a target.
3. **Runtime logging.** A Frida framework that logs the live IL2CPP class graph and traces outbound synchronization traffic.
4. **Attack hooks.** The main instrumentation methods used and sanitized example hooks.
5. **Helper scripts.** Helper scripts, a Mono runtime bridge for Unity clients that do not use the IL2CPP backend, and verification scripts that check the static-analysis output.

A.1.2 Private Component

For ethical reasons (responsible disclosure), the operational material that weaponizes these tools (i.e., symbol offsets, ready-to-run per-app hooks, the modified C++ client, and the target binaries) is withheld from the public release, as stated in our paper's Open Science statement. It was shared confidentially with the Artifact Evaluation Committee for the reproducibility assessment. It is available to researchers on request.

A.2 Description & Requirements

A.2.1 Security, privacy, and ethical concerns

Following responsible disclosure, the operational material that could enable misuse (the app-specific hooks, symbol offsets, the modified client, and the target binaries) is withheld from the public release and provided only to the artifact evaluators for reproducibility assessment, so that it cannot be misused before the vendor fixes ship. The work involved no human subjects: all demonstrations used our own test accounts in private sessions. The artifact does not contain any destructive components, so it is safe to run the experiments. The instrumentation is intended for use only against systems and accounts you own or are explicitly authorized to test.

A.2.2 How to access

Public Component. Zenodo concept DOI:

`https://doi.org/10.5281/zenodo.20434355`

The record publishes two files: `relay-and-betray.zip` (the complete toolchain and a stage-by-stage README) and `videos.zip` (the demonstration videos).

Private component (confidential; shared only with AEC). It contains the app-specific hooks, the target binaries, modified APKs, and sample static analysis outputs. Access was granted to the AEC for the duration of the evaluation; that credential has since been revoked, as this material cannot be released publicly for ethical reasons. Consistent with the paper's Open Science statement, we may make it available to other researchers on request, after verification on a case-by-case basis.

A.2.3 Hardware dependencies

Static analysis (Il2CppDumper, Ghidra) runs on any Windows 11, macOS, or Linux machine with a multi-core CPU, $\geq$ 16,GB RAM, and $\sim$30,GB free disk.

Gadget injection runs on macOS or Linux, and additionally needs `adb` and the target Android-based device attached over USB with developer mode enabled.

Runtime logging and attack hooks require running the application on any supported platform. A VR headset (e.g., Quest, Vive) is optional for apps available on PC VR as they can be launched in desktop mode. (Rendering an MR client in desktop mode needs a GPU with more than 4,GB of VRAM.) A GUI is necessary for reproducing results because every attack is confirmed perceptually. You need a second client on a separate endpoint (e.g., headset, PC, smartphone) to verify the attacks.

A.2.4 Software dependencies

Vanilla Windows 11, macOS, or Linux. `setup.sh` / `setup.ps1` install the pinned open-source dependencies: Frida 17.9.1, Node.js, Python 3.12, JDK 21 (for the bundled Ghidra 11.3.2), and the .NET 6 runtime (for the bundled Il2CppDumper); `adb` is needed only for the device path. Running both clients on a single machine additionally needs a sandboxing tool such as Sandboxie-Plus.

A.2.5 Benchmarks

None.

A.3 Set-up

A.3.1 Installation

1. Download both files from the Zenodo DOI and run the setup script (`setup.sh` on macOS/Linux, `setup.ps1` on Windows) to install all dependencies. On Windows, `setup.ps1 -StaticAnalysis` also prepares the JDK and .NET runtime that the bundled Ghidra and Il2CppDumper need.

2. Install the target app and create two free test accounts (one per client).

3. For (E3) only: obtain the private component as described in § A.2.2.

The README walks through the pipeline as ordered stages, each one command with the output to expect: install, obtain the app binary, recover symbols, decompile, verify, log at runtime, and run a hook. It also works one sanitized example hook end to end, showing how the runtime logger supplies the values that were blanked out.

A.3.2 Basic Test

`frida -version` prints `17.9.1`, and running the dumper on an input binary produces a non-empty `dump.cs` (recovered classes). Together these confirm that both halves of the toolchain work.

A.4 Evaluation workflow

A.4.1 Major Claims

(C1) The static-analysis pipeline deterministically recovers the class, method, and field symbols the attacks rely on, from a stripped Unity IL2CPP binary, with no headset required. This runs with the public component alone. The input binary is supplied by the user, extracted from their own installed app as the README explains, or taken from the sample inputs in the private component; reference output is included there for direct comparison. Proven by experiment (E1).

(C2) Using the operational hooks in the private component against a live app, each of the five invariant violations—perceptual secrecy, state integrity, safety enforcement, availability, and identity integrity—is reproducible across select apps. The attacks need no headset and run on a standard Windows machine with any commercial GPU (more than 4 GB of VRAM). Proven by experiments (E2) and (E3).

We deliberately bound what C2 covers to limit the specialized hardware needed to execute and verify the attacks: the hooks we share target apps that are available across multiple platforms and whose modified client runs on a standard Windows machine. Because these apps' code keeps evolving—and so do the hooks, especially as we coordinate fixes for the disclosed vulnerabilities—and because each attack on each app requires perceptual verification, we provide hooks and step-by-step instructions for a selected set of apps that establish the paper's key claims, rather than for all 20.

A.4.2 Experiments

(E1): *Symbol recovery and decompilation* [$\sim$15 human-min + $\sim$2 compute-hours]. Run the dumper on the input binaries (`libil2cpp.so` and `global-metadata.dat`), then import the recovered symbols in headless Ghidra, which applies them and exports decompiled C per class. The result is a non-empty symbol dump (`dump.cs`) listing recovered classes, methods, and fields, and a per-class export in which functions carry real names rather than `FUN_xxxx`. Il2CppDumper is deterministic, so the same binary yields the same dump every run; Ghidra is not, as its decompiler makes heuristic choices about naming and typing, so spelling and formatting vary between runs while the recovered structure and logic stay comparable. Verification scripts (`verify.ps1`, `verify.sh`) check both stages and print PASS/FAIL; the output can also be inspected in the Ghidra GUI, or compared against the reference output included in the private component.

(E2): *Runtime logging* [∼15 human-min]. Attach the Frida logger to the running client to enumerate its class graph and locate the methods each attack targets. The printed inventory also identifies the networking and voice SDK in use and the outbound event codes, which are how the placeholders in the sanitized public hooks are filled in.

(E3): *Attack confirmation* [∼2 human-hours]. Launch an attacker client and a victim client in the same private room, making the victim the room host for the host-control attacks. For each attack, load the runtime bridge first, then the corresponding hook; the attach helper does both and identifies the attacker instance automatically. Each attack produces its stated effect on the victim: perception leaking past secrecy settings, unauthorized state or object changes, a non-host exercising host-only controls, a forced crash or disconnection, or impersonation under another user's identity. The demonstration videos in `videos.zip` show the expected outcome for each attack.

A.5 Notes on Reusability

The public component is written to be reused against apps beyond the 20 we studied. The static-analysis pipeline and the runtime logger take no app-specific input: given any Unity IL2CPP client, they recover its symbol map and print its live class graph, networking SDK, and synchronization traffic. The gadget-injection toolchain handles split APKs, license checks, and anti-debug checks generically, so it applies to non-rooted Quest and Vive targets in general. Not every Unity MR client uses the IL2CPP backend, and frida-il2cpp-bridge cannot attach to the ones that do not, so we also ship a Mono runtime bridge exposing an equivalent API; the same hooks therefore apply to either scripting backend. The instrumentation methods in `hook_patterns.js`—modify arguments, modify the return value, read or write fields, and override method logic—are the primitives all of our attacks are built from, and the sanitized examples show how each invariant violation is expressed in those terms. Applying the invariant-guided methodology to a new or updated MR app therefore does not require anything we withheld.

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/usenixsec2026/.